Cover Page

Universiteit Leiden

Author: Chaudron, Michel
Title: Separating computation and coordination in the design of parallel and distributed programs
Issue Date: 1998-05-28

Separating Computation and Coordination in the Design of Parallel and Distributed Programs

Cover by Victor Vasarely, 1979

Separating Computation and Coordination in the Design of Parallel and Distributed Programs

PROEFSCHRIFT

Ter verkrijging van de graad van Doctor
aan de Rijksuniversiteit te Leiden,
op gezag van de Rector Magnificus Dr. W. A. Wagenaar,
hoogleraar in de faculteit der Sociale Wetenschappen,
volgens besluit van het College voor Promoties
te verdedigen op donderdag 28 Mei 1998
te klokke 14.15 uur

door

Michel Roger Vincent Chaudron
geboren te Leiden in 1969

samenstelling van de promotiecommisie

promotor:	Prof. dr. F. J. Peters	
co-promotor:	Dr. E. de Jong	Hollandse Signaalapparaten B.V.
referent:	Prof. dr. C. L. Hankin	Imperial College of Science, Technology and Medicine, London, Engeland
overige leden:	Prof. dr. J. W. de Bakker	Centrum voor Wiskunde en Informatica, Amsterdam
	Prof. dr. J. N. Kok	
	Prof. dr. G. Rozenberg	
	Prof. dr. H. A. G. Wijshoff	

This work was carried out in graduate school ASCI.
ASCI dissertation series number 31.

Separating Computation and Coordination in the Design of Parallel and Distributed Programs Michel Roger Vincent Chaudron. - [S.l. : s.n.].-Ill.
Thesis Rijksuniversiteit Leiden. - With ref.
ISBN 90-9011643-5
NUGI 851
Subject headings: coordination / formal methods / parallel computing

Aan mijn ouders

Contents

1 Introduction

Computers are so widely used throughout society that they affect our lives 24 hours a day. They are the culmination of a tradition of making of tools to aid us in our daily lives. The successful application of computers derives from two of their abilities:

- performing computations at a phenomenal speed (currently many millions per second), and
- storing tremendous amounts of data (currently in the order of one terra-byte).

Both of these abilities can be further increased by coupling computers. Two computers can, in principle, compute twice as fast and store twice as much as a single computer.

Although magical qualities are sometimes attributed to computers, they are but machines whose task it is to mechanically carry out our instruction. They are, however, very complex machines.

Software is as complex a construction as hardware. Several layers of software are used to steer the operation of the hardware. Each layer of software provides (suitable) abstractions for a layer one level up in the hierarchy, thereby hiding more and more machine specific aspects. The top most layer consists of an application which provides the functionality presented to the users of a system. Application programs that consist of millions of lines of code are no exception.

The building of such programs is a formidable task, which, unfortunately, is very error-prone. For software engineers it would be desirable if they could employ methods which would help in making error-free software products. This raises questions regarding the principles on which such methods could be based.

For traditional engineering products, it is common practice to expose products to extensive testing. Mechanical constructions may be tested under extreme circumstances, such as extreme temperature or force. Conclusions are then drawn from assumptions about continuity of behaviour for intermediate circumstances.

However, this assumption of continuity does not carry over to computer programs. Due to their discrete nature, a difference of a single bit or instruction may cause a

program to fail or deviate from its intended behaviour in a completely unexpected way. Hence, to ascertain the correctness of a system, every individual setting of bits and every possible sequence of instructions would need to be verified. The number of possible settings and sequences is of such cosmic proportions that even computers themselves cannot help us in checking all possibilities.

Hence, the method of testing cannot be used to guarantee the correct operation of computer systems. The only way to assert anything about the correctness of a design is by rigorously specifying the system requirements such that mathematical methods of reasoning can be applied.

The question whether the initial specifications of a system actually meets the requirements is impossible to answer, since there is inherent ambiguity in the formalization of an informal problem statement. However, a formal specification already provides an advantage in that it can be checked on internal consistency. Furthermore, we may subsequently appeal to mathematical methods to assist us in the transformation of an initial specification into more and more detailed form up to the stage where it becomes computer-processable.

Rather than going through the trouble of instructing computers in every detail, we can also try to raise the level of abstraction at which computers can execute our instructions. Programming languages and tools should support the programming activity by assisting us in focusing on the relevant aspects at different stages of the design process and encourage abstraction from details that should be addressed at a different stage or from details that can be resolved automatically. One successful step in this direction has been the use of compilers to translate (so-called) high level languages into machine-oriented instructions.

With the advent of parallel and distributed computer systems, the complexity of designing programs was increased even further by the need to resolve matters of concurrency and distribution. In order to deal with these matters, programming languages were extended with new primitives that were tailored for explicitly defining communication and distribution aspects. This increased the complexity of the programming languages and the programming activity because it encouraged the programmer to think about functional and operational issues at the same time.

The principle of "separation of concerns" (see e.g. [61]) has been proposed as a means of structuring the design of programs. This principle identifies correctness and complexity as the two main concerns of a programmer. The method of design associated with this principle consists of first concentrating on the functionality while abstracting

from operational issues. Subsequently, a program may be further refined, aimed at improving the efficiency while leaving the functionality unaffected.

Up until now, formal methods could be used in support of this approach towards program design, but no formal methods have been put forward that encourage (or enforce) the separation of concerns. The research described in this thesis aims to fill this gap.

In this thesis, we propose a collection of formal techniques that constitute a methodology for the design of parallel and distributed programs which addresses the correctness and complexity aspects in separate phases. This method proceeds along the following phases. For each phase, we present formalisms for specifying and reasoning about the aspects that belong to that each phase and encourage abstraction from aspects that belong to the complementary phase.

Firstly, we concentrate on specifying the functionality (which defines "what" should be computed). This specification determines the correctness of the program and is called the "computation component". In support of this phase we present a programming model which allows the description of the basic computations that constitute a solution method, but abstracts from an underlying execution mechanism and thereby avoids to impose premature constraints on the order of computation. This programming model is a variant of Gamma [12, 13] and the Transaction-based programming model [79]. However, in order to incorporate it in our formal methodology, we provided it with an alternative formal semantics.

Secondly, we concentrate on specifying "how" a program should operate in order to compute what its functional component promises. This specification is called the "coordination component" and it complements the computation component by defining the operational aspects of a system. For instance, depending on the target architecture, the coordination component may prescribe a sequential or parallel strategy for realising the computation component.

In support of this second phase we develop a coordination language which is aimed at describing the behaviour of a system in terms of the basic computations of the solution method. We formally define the syntax and semantics of this language such that it can be integrated in our formal methodology.

To ensure the correctness of coordination components, we construct a formal method for their development by stepwise refinement.

The stepwise development of coordination components is put on a formal basis by developing a collection of formal methods for refinement which allow a modular, transformational manner of reasoning.

Structure of this thesis

The remainder of this thesis is organized as follows. Chapters 2 and 3 introduce the specification formalisms that are used in this thesis. In Chapter 2 we present the computation language. We show that it facilitates the description of specifications that are not partial to a particular mode of execution. Furthermore, we present a semantics and a logic for reasoning about correctness of programs. In Chapter 3 we present the coordination language. We define its semantics and show how it connects to the computation language.

In Chapters 4 and 5 we develop a theory of refinement. This theory provides a number of proof techniques that enable us to incrementally refine the behavioural aspects of a program. These chapters form the most theoretical part of this thesis. It should be possible to get an understanding of the methods derived in these chapters without going through all these proofs.

In Chapter 7 we illustrate the method of design by considering some case studies. Comparisons with related work and conclusions are described in Chapters 8 and 9.

2 The Computational Model

The aim of this thesis is to develop a methodology which supports the separate design of the computation and coordination aspects of programs. In order to realize this approach, we need a programming model that supports this separation between computation and coordination. Existing programming models usually stress only one of these aspects. For instance, functional and logical programming languages emphasize their declarative nature and the advantages it has for proving correctness, but deny the programmer effective means for determining the program's behaviour. With imperative languages the programmer has complete control over the operational behaviour of his program. However, because the control-flow is an integral part of imperative programs, it is difficult to focus on correctness while abstracting from operational details.

In this chapter we will present the Gamma programming model which has shown to be well suited for specifying the computation component of a program without imposing premature constraints on the coordination component. We present a concise semantics for Gamma programs and a formal logic for reasoning about their correctness.

2.1 The Gamma Programming Model

We start with a brief introduction to Gamma. For more details the reader is referred to [13] which includes a series of example programs.

The uniform data structure in Gamma is the multiset. Multisets can be formed over arbitrary domains of values, including integers, reals, booleans and tuples. The simplest Gamma program is a conditional multiset rewrite-rule, written as $\overline{x} \mapsto m \Leftarrow b$. Here $\overline{x}$ denotes a sequence of variables $x_1, \ldots, x_n$, m is a multiset expression, and b is a boolean expression. The free variables that occur in m and b are taken from $x_1, \ldots, x_n$.

Application of the rule $\overline{x} \mapsto m \Leftarrow b$ to a multiset proceeds by replacing elements in the multiset satisfying the condition b by the elements that result from evaluating the multiset expression m. This step is repeated until no more elements are present that

satisfy b. The resulting multiset denotes the outcome of the program.

Example 2.1.1 *We introduce a Gamma program for sorting a sequence of numbers into ascending order. The input sequence is represented by a multiset consisting of value-index pairs. The Gamma program consists of a single multiset rewrite rule which is defined as follows*

$$swap \mathrel{\widehat{=}} (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow x > y \ \wedge \ i < j \tag{2.1}$$

The program executes by exchanging ill-ordered values until there are no more pairs that satisfy this condition. At that point the resulting multiset represents a well-ordered sequence. Disjoint pairs can be compared and exchanged in parallel, but this need not necessarily be the case.

A possible execution for the sorting program is depicted in Figure 2.1.

Figure 2.1: Possible execution of the program swap *in a multiset* $M_0 = \{(1,D),(2,C),(3,B),(4,A)\}$

It is important to note that the Gamma program does not specify in which order pairs of values are compared and exchanged. Hence the program can be seen as a highly

nondeterministic specification of a wide spectrum of sorting strategies. This opens up the opportunity for a separate specification of the operational aspects of the program. This gap will be filled by the coordination language that we will present in Chapter 3.

More complex Gamma programs can be built using two basic combinators. Individual rules can be composed into so-called *simple* programs [65] using the parallel combinator, denoted "+". The constituent rules in parallel composition are executed in any order (possibly in parallel) until none of the rules can be successfully applied.

Simple programs can in turn be composed using the sequential combinator , denoted " $\circ$ ". If P_1 and P_2 are simple programs, then $P_1 \circ P_2$ first executes P_2 until its rules can no longer be applied, after which P_1 is executed on the resulting multiset.

The abstract syntax of Gamma programs can be specified as follows. We use r, R and P to range over the syntactic categories of multiset rewrite-rules, simple programs, and programs respectively. We use $\mathbb{P}$ to denote the set of all Gamma programs.

Syntactic Categories

r	$\in$	*Rule*
R	$\in$	*Simple*
P	$\in$	*Program*

Definition

r	$::=$	$\overline{x} \mapsto m \Leftarrow b$		
R	$::=$	r	$\mid$	$R + R$
P	$::=$	R	$\mid$	$P \circ P$

Figure 2.2: Abstract Syntax of Multiset Transformer Programs

The program terms derivable in this way are "products of sums"; i.e. are of the form $(r_1 + \cdots + r_i) \circ \cdots \circ (r_j + \cdots + r_n)$. The purpose of limiting the syntax of program terms to this form is to exclude the parallel composition of programs that contain sequential composition; e.g. $P_1 + (P_2 \circ P_3)$. There are two reasons for excluding these forms: firstly, the syntax thus obtained describes exactly the same set of programs that are definable by the original Gamma model presented in [12] and [13]. Secondly, the excluded terms present difficulties with the compositionality of semantics [34]. An additional justification for the focus on programs in product-of-sums form is a result of Sands [106] which entails that every Gamma program can be refined by a program that is in product-of-sums form.

We use the method of *structural operational semantics* [96] to define the meaning of Gamma programs. To this end we introduce configurations, denoted $\langle P, M\rangle$, where P is a Gamma program and M is a multiset. A configuration represents the state of a computation. A configuration can move to another configuration by performing an action. Such actions are modelled by a relation between configurations. To define the semantics of Gamma, we use a labelled multi-step transition relation.

A transition is written as $\langle P, M\rangle \overset{\sigma}{\rightsquigarrow} \langle P', M'\rangle$ where the label σ is a multiset substitution which formally describes the rewrite action that transforms M into M'. A terminal configuration is written $\langle P, M\rangle\surd$.

The semantics of Gamma is collected in Figure 2.3. The multi-step transition relation is defined in terms of a single-step transition relation. The latter is distinguished from the multi-step transition relation by decorating it with a subscript "1": $\overset{\sigma}{\rightsquigarrow}_1$. This single-step transition relation will be used in Chapter 3 to link the semantics of the coordination component to that of the computation component.

The various notations that we use in defining the semantics are best explained by considering the semantic rule for execution of an individual rewrite rule $r = \overline{x} \mapsto m \Leftarrow b$:

$$\text{if } \overline{v} \subseteq M : b[\overline{x} := \overline{v}] \text{ then } \langle r, M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle r, M[\sigma]\rangle \text{ where } \sigma = m[\overline{x} := \overline{v}]/\overline{v}$$

We write $b[\overline{x} := \overline{v}]$ to denote the boolean expression that results from replacing each free occurrence of x_i by v_i. We write $\sigma = M/N$ to denote a multiset substitution σ which replaces N by M. By $M[\sigma]$ we denote the multiset that results from applying the substitution σ to M. More formally, let $M' = m[\overline{x} := \overline{v}]$, then $M[M'/\overline{v}] = (M \ominus \overline{v}) \oplus M'$, where $\oplus$ and $\ominus$ denote multiset addition and subtraction respectively (their formal definition can be found in Appendix A.2). Note that for ease of notation we confuse the sequence $\overline{v}$ with the multiset consisting of the same elements as $\overline{v}$.

When multiple transitions transform disjoint parts of the multiset, then these transitions do not interfere with each other, hence they can also happen in parallel. This observation directly leads to the multi-step transition semantics of Gamma, in particular semantic rule (C4), as defined in Figure 2.3.

We present two variants of a formal definition of non-interference . The first notion is the most strict: it requires the elements that are retrieved from the multiset to be strictly disjoint. The second is more flexible: it allows elements to be read[1] concurrently by multiple multiset substitutions. This difference corresponds to *exclusive read/exclusive write*

[1]The removal and insertion of identical elements by a single multiset-substitution is interpreted as reading of those elements.

and *concurrent read/exclusive write* mechanisms found in the classification of architectures for parallel computers. By default, we use Definition 2.1.3.

Definition 2.1.2 *Given a multiset M and two multiset substitutions $\sigma_1 = M_1/N_1$ and $\sigma_2 = M_2/N_2$, we say that σ_1 and σ_2 are* independent *in M, denoted $M \models \sigma_1 \bowtie_E \sigma_2$, if $N_1 \oplus N_2 \subseteq M$.*

Definition 2.1.3 *Given a multiset M and two multiset substitutions $\sigma_1 = M_1/N_1$ and $\sigma_2 = M_2/N_2$.*

1. *We say that σ_1 is* independent from *σ_2 in M, denoted $M \models \sigma_1 \triangleleft \sigma_2$, if $N_1 \subseteq (M \ominus N_2) \cup M_2$.*
2. *We write $M \models \sigma_1 \bowtie \sigma_2$ if σ_1 and σ_2 are* mutually independent *in M; i.e. if $M \models \sigma_1 \triangleleft \sigma_2$ and $M \models \sigma_2 \triangleleft \sigma_1$*

The label assigned to a multi-step transition is a combination of the labels of the constituent transitions. The concurrence of multiple multiset substitutions can be formally described using the composition operator.

Definition 2.1.4 *Given two multiset substitutions $\sigma_1 = M_1/N_1$ and $\sigma_2 = M_2/N_2$, the* composition *of σ_1 and σ_2 is defined as $\sigma_1 \cdot \sigma_2 = (M_1 \oplus M_2)/(N_1 \oplus N_2)$.*

(C0) $$\langle \overline{x} \mapsto m \Leftarrow b, M\rangle\surd \qquad \text{if } \neg(\exists \overline{v} \subseteq M : b[\overline{x} := \overline{v}])$$

(C1) $$\langle \overline{x} \mapsto m \Leftarrow b, M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle \overline{x} \mapsto m \Leftarrow b, M[\sigma]\rangle \qquad \text{if } \overline{v} \subseteq M \wedge b[\overline{x} := \overline{v}] \text{ where } \sigma = m[\overline{x} := \overline{v}]/\overline{v}$$

(C2) $$\frac{\langle R, M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle R, M'\rangle}{\langle R, M\rangle \overset{\sigma}{\rightsquigarrow} \langle R, M'\rangle}$$

(C3) $$\frac{\langle R_1, M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle R_1, M'\rangle}{\begin{array}{c}\langle R_1 + R_2, M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle R_1 + R_2, M'\rangle \\ \langle R_2 + R_1, M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle R_2 + R_1, M'\rangle\end{array}}$$

(C4) $$\frac{\begin{array}{c}\langle R, M\rangle \overset{\sigma_1}{\rightsquigarrow}_1 \langle R, M_1\rangle \\ \langle R, M\rangle \overset{\sigma_2}{\rightsquigarrow} \langle R, M_2\rangle\end{array}}{\langle R, M\rangle \overset{\sigma_1 \cdot \sigma_2}{\rightsquigarrow} \langle R, M[\sigma_1 \cdot \sigma_2]\rangle} \qquad \text{if } M \models \sigma_1 \bowtie \sigma_2$$

(C5) $$\frac{\begin{array}{c}\langle R_1, M\rangle\surd \\ \langle R_2, M\rangle\surd\end{array}}{\langle R_1 + R_2, M\rangle\surd}$$

(C6) $$\frac{\langle P_1, M\rangle\surd \quad \langle P_2, M\rangle \overset{\sigma}{\rightsquigarrow} \langle P_2', M'\rangle}{\langle P_2 \circ P_1, M\rangle \overset{\sigma}{\rightsquigarrow} \langle P_2', M'\rangle}$$

(C7) $$\frac{\langle P_1, M\rangle \overset{\sigma}{\rightsquigarrow} \langle P_1', M'\rangle}{\langle P_2 \circ P_1, M\rangle \overset{\sigma}{\rightsquigarrow} \langle P_2 \circ P_1', M'\rangle}$$

(C8) $$\frac{\begin{array}{c}\langle P_1, M\rangle\surd \\ \langle P_2, M\rangle\surd\end{array}}{\langle P_1 \circ P_2, M\rangle\surd}$$

Figure 2.3: Semantics of Multiset Transformer Programs

The semantics of Gamma as defined in Figure 2.3 differs from the one presented in [65]. The latter uses a single-step transition relation which suggests an interleaved semantics.

The idea behind the coordination language that we will present in Chapter 3 is that it restricts the otherwise nondeterministic behaviour of Gamma programs, hence it cannot introduce new behaviour. Consequently, the semantics we choose for programs, limits the behaviours we can obtain using a coordination language. Because we want to distinguish between parallel and sequential execution at the coordination level, we need this distinction to be present in the semantics of Gamma.

In Section 9.2.3 of Chapter 9 we will describe a technical anomaly of single-step semantics that occurs in the context of refinement. The fact that multi-step semantics does not exhibit this anomaly is another reason for preferring it over single-step semantics.

The multi-step operational semantics of Figure 2.3 and the single-step operational semantics of [65] endow Gamma programs with different behaviour (in the sense of the possible (sequences of) transitions), but induce the same functionality (input-output relation) for programs.

To prove the functional equivalence between the multi-step and single-step semantics, we formalize the notion of input-output relation. To this end, we first define the reflexive transitive closure of the transition relation and a "may diverge" predicate.

Definition 2.1.5 *We define the reflexive transitive closure of the transition relation, denoted $\rightsquigarrow^*$, by*

$$\langle P, M\rangle \overset{\langle\,\rangle}{\rightsquigarrow}{}^*\langle P, M\rangle \qquad \frac{\langle P, M\rangle \overset{\sigma}{\rightsquigarrow} \langle P', M'\rangle}{\langle P, M\rangle \overset{\sigma}{\rightsquigarrow}{}^*\langle P', M'\rangle} \qquad \frac{\langle P, M\rangle \overset{\overline{\sigma_1}}{\rightsquigarrow}{}^*\langle P', M'\rangle \quad \langle P', M'\rangle \overset{\overline{\sigma_2}}{\rightsquigarrow}{}^*\langle P'', M''\rangle}{\langle P, M\rangle \overset{\overline{\sigma_1}\cdot\overline{\sigma_2}}{\rightsquigarrow}{}^*\langle P'', M''\rangle}$$

The reflexive transitive transition relation uses labels $\overline{\sigma}$ which denote sequences of individual labels. For convenience we identify the singleton sequence $\langle\,\sigma\,\rangle$ with its only element σ.

Definition 2.1.6 *A configuration $\langle P, M\rangle$ may diverge, denoted $\langle P, M\rangle\uparrow$, if and only if $\langle P, M\rangle = \langle P_0, M_0\rangle$ and for all $i \geq 0$ there exists a σ_i such that $\langle P_i, M_i\rangle \xrightarrow{\sigma_i} \langle P_{i+1}, M_{i+1}\rangle$.*

Definition 2.1.7 *The capability function for programs $\mathcal{C} : \mathbb{P} \times \mathbb{M} \to \mathcal{P}(\mathbb{M}) \cup \{\bot\}$ is*

defined as

$$\mathcal{C}(P,M) = \ \{\bot \mid \langle P,M\rangle\!\uparrow\} \ \cup \ \{M' \mid \langle P,M\rangle \overset{\bar{\sigma}}{\rightsquigarrow}{}^{*} \langle P',M'\rangle\surd\}$$

Example 2.1.8 *Consider the sorting program from Example 2.1.1 and an initial sequence* $\langle\, 13,7,97\,\rangle$*. Then* $\mathcal{C}(swap,\{(1,13),(2,7),(3,97)\}) = \{\{(1,7),(2,13),(3,97)\}\}$.

We show the functional equivalence of the multi-step and single-step semantics for simple programs. The generalization to arbitrary Gamma programs is straightforward.

The multi-step semantics for simple Gamma program consists of the inference rules (C0), (C1), (C2), (C3), (C4) and (C5) from Figure 2.3. The single-step semantics consist of inference rules (C0), (C1), (C2), (C3) and (C5). We use $\mathcal{C}_1$ to denote the capability function for the single-step semantics.

First, we show that for every multi-step transition there exists a sequence of single-step transitions, denote $\rightsquigarrow_1{}^{*}$, that has the same effect on the multiset.

Many of the lemmas in this thesis, for example Lemma 2.1.9, are of the form "if some transition $\langle P,M\rangle \overset{\sigma}{\rightsquigarrow} \langle P',M'\rangle$, then some conclusion". The method of structural operational semantics [96] ensures that every transition is derived by a finite number of inferences using the semantic rules. This allows statements of the aforementioned type, to be proven by induction on the depth of this inference tree. This method is called *proof by transition induction* . This technique is used the proof of Lemma 2.1.9.

Lemma 2.1.9 *Let* P *be a simple program. If* $\langle P,M\rangle \overset{\sigma}{\rightsquigarrow} \langle P,M'\rangle$*, then there exists a sequence of single-step transitions*

$$\langle P,M_0\rangle \overset{\sigma_1}{\rightsquigarrow}_1 \langle P,M_1\rangle \ldots \overset{\sigma_i}{\rightsquigarrow}_1 \ldots \langle P,M_{n-1}\rangle \overset{\sigma_n}{\rightsquigarrow}_1 \langle P,M_n\rangle$$

such that $M_0 = M$ *and* $M_n = M'$ *and* $\sigma = \sigma_1 \cdot \ldots \cdot \sigma_n$.

Proof By transition induction: consider the possible ways in which the last inference of the transition $\langle P,M\rangle \overset{\sigma}{\rightsquigarrow} \langle P,M'\rangle$ may have been made.

- by (C2) from $\langle P,M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle P,M'\rangle$. Then the result holds directly.
- by (C4) from $\langle P,M\rangle \overset{\sigma_1}{\rightsquigarrow}_1 \langle P,M''\rangle$ and $\langle P,M\rangle \overset{\sigma_2}{\rightsquigarrow} \langle P,M'''\rangle$ where $M \models \sigma_1 \bowtie \sigma_2$. From Lemma A.2.6 follows that these transitions may be applied in arbitrary interleaved order; for instance

$$\langle P,M\rangle \overset{\sigma_1}{\rightsquigarrow}_1 \langle P,M''\rangle \quad \text{and} \quad \langle P,M''\rangle \overset{\sigma_2}{\rightsquigarrow} \langle P,M'\rangle$$

By the induction hypothesis we get for the latter transition that there exists a sequence of single-step transitions

$$\langle P, M_0\rangle \overset{\sigma'_1}{\rightsquigarrow}_1 \langle P, M_1\rangle \ldots \overset{\sigma'_i}{\rightsquigarrow}_1 \ldots \langle P, M_{n-1}\rangle \overset{\sigma'_n}{\rightsquigarrow}_1 \langle P, M_n\rangle$$

such that $M_0 = M''$ and $M_n = M'''$ and $\sigma_2 = \sigma'_1 \cdot \ldots \cdot \sigma'_n$. The result follows from concatenation of this sequence of single-step transitions to $\langle P, M\rangle \overset{\sigma_1}{\rightsquigarrow}_1 \langle P, M''\rangle$.

□

Theorem 2.1.10 *Let P be a simple program. Then, $\forall M : \mathcal{C}(P, M) = \mathcal{C}_1(P, M)$.*

Proof We prove that $\mathcal{C}(P, M) \subseteq \mathcal{C}_1(P, M)$ and $\mathcal{C}_1(P, M) \subseteq \mathcal{C}(P, M)$.

- $\mathcal{C}(P, M) \subseteq \mathcal{C}_1(P, M)$: By Lemma 2.1.9 follows that every multi-step transition can be mimicked by a sequence of single-step transitions. By induction on the length of the transition sequence follows that the single-step semantics can mimic any sequence of multi-step transitions (be it a finite or infinite sequence).

- $\mathcal{C}_1(P, M) \subseteq \mathcal{C}(P, M)$: This follows from the fact that the inference rules for the single-step semantics are a subset of the inference rules for the multi-step semantics.

□

In the next section we will present a formal method for reasoning about Gamma programs.

2.2 Reasoning about Gamma Programs

In this section we briefly introduce a method for reasoning about Gamma programs that is inspired on the UNITY logic [23] and its application to multiset transformer programming as first described in [79]. This method complements the methods proposed in [12] in that it is suitable for the a-posteriori verification of programs. Furthermore, it enables us to establish properties of Gamma programs that we can exploit in later stages of development where we concentrate on refinement of coordination strategies for Gamma programs.

We introduce a small repertoire of basic properties that suffices for the applications in this thesis. It is straightforward to extend this repertoire with other properties (such as appear in UNITY [23] or other temporal logics).

We write quantified predicates on multisets in the following way.

$$[\![\textit{quantifier variable-list} : \textit{range-expression} : \textit{boolean-expression}]\!] \textit{multiset}$$

The variables that occur in the *variable-list* range over all values in the *range-expression* that are present in a given multiset.

$$\begin{array}{lcl} [\![\forall \overline{x} : ran(\overline{x}) : p]\!]M & \Leftrightarrow & \forall \overline{v} : \overline{v} \subseteq M \wedge ran(\overline{v}) : p[\overline{x} := \overline{v}] \\ [\![\exists \overline{x} : ran(\overline{x}) : p]\!]M & \Leftrightarrow & \exists \overline{v} : \overline{v} \subseteq M \wedge ran(\overline{v}) : p[\overline{x} := \overline{v}] \end{array}$$

For example, $[\![\forall s, i, x_i : (\mathcal{X}, s, i, x_i) : s \geq 0]\!]M$ should be read as: 'for all values s, i, x_i such that there is a tuple $(\mathcal{X}, s, i, x_i)$ in multiset M, holds that s is greater than or equal to zero".

Following [23] we also use quantified expressions (over multisets) where a binary, associative and commutative operator with a unit element is used instead of a quantifier[2].

$$[\![\textit{ operator variable-list} : \textit{range-expression} : \textit{numerical-expression}]\!] \textit{ multiset}$$

For example, $[\![+ t, i, z : (\mathcal{Z}, t, i, z) : t]\!]M$ denotes the sum of all values t for which there is a tuple $(\mathcal{Z}, t, i, z)$ for some t, i and i in multiset M. If the range of the quantification is empty, then the value of the expression is the unit element of the operator. The unit elements of *min*, *max*, $+$, $*$ are ∞, $-\infty$, 0 and 1 respectively.

In addition, we use the symbol '#' as a counting quantifier (as introduced by [61]). Formally,

$$[\![\#x : p]\!]M \;=\; \Sigma_{a \in A} f(a) \text{ where } f(a) = \begin{cases} M(a) & \text{if } p[x := a] \\ 0 & \text{otherwise} \end{cases}$$

For example, $[\![(\#s, i, x : (\mathcal{X}, s, i, x)) : \mathit{true}]\!]M$ denotes the number of tuples of the form $(\mathcal{X}, s, i, x)$ in the multiset M.

In contrast to [23], we define the properties of our logic in terms of the formal operational semantics of programs (Figure 2.3). Let q, q' be quantified predicates on multisets,

[2]Although this notation is a debatable deviation from the mathematical convention, it constitutes a uniform notation and avoids long subscripts (especially with $\sum$ or $\prod$) which regularly occur when working with tuples rather than numbers.

let M_i, M' etc. denote multisets. Let $\langle P, M_0\rangle$ be the initial configuration of some program P.

- *initially* q iff $[\![q]\!]M_0$
- q *unless* q' iff

$$(\forall P', P'', M', M'' : \langle P, M_0\rangle \rightsquigarrow^* \langle P', M'\rangle \rightsquigarrow \langle P'', M''\rangle : ([\![q \wedge \neg q']\!]M' \Rightarrow [\![q \vee q']\!]M'')$$

  From an operational point of view, q *unless* q' means that if q holds at some point of the computation, and q' does not, then after the next transition, either q continues to hold or q' starts to hold.

- *stable* q iff q *unless false*
  A predicate q is stable, if, once predicate q holds at some point of the computation, it will continue to hold. However, q may never start to hold.

- *invariant* q iff *initially* $q \wedge$ *stable* q
  A invariant q is a stable predicate that holds throughout the computation.

In addition to these, we introduce the termination condition, denoted $\dagger P$, that characterizes the final state(s) of a (simple) program. Enabledness of a (simple) program, which is dual to termination, is denoted $\natural P$.

$$\begin{array}{lcl} [\![\natural(r_1 + \ldots + r_n)]\!]M & \Leftrightarrow & \exists i : 1 \leq i \leq n : (\exists \overline{v} : \overline{v} \subseteq M : b_i[\overline{x} := \overline{v}]) \\ [\![\dagger(r_1 + \ldots + r_n)]\!]M & \Leftrightarrow & \forall i : 1 \leq i \leq n : (\forall \overline{v} : \overline{v} \subseteq M : \neg b_i[\overline{x} := \overline{v}]) \end{array}$$

The termination condition of a program can be derived in a syntactical manner by negating the conditions of the rewrite rules that constitute the program.

Lemma 2.2.1 *Let $P = r_1 + \ldots + r_n$ be a simple program.*
If $\langle P, M\rangle\surd$, then $\forall i : 1 \leq i \leq n : [\![\dagger r_i]\!]M$

Proof By transition induction using the semantics from Figure 2.3 follows that $\langle P, M\rangle\surd \Leftrightarrow (\forall i : 1 \leq i \leq n : \langle r_i, M\rangle\surd)$. The result follows from $\langle r_i, M\rangle\surd \Leftrightarrow [\![\dagger r_i]\!]M$. □

A specification of the desired output of a program is called that program's postcondition. The postcondition of Gamma programs can be specified using the quantified predicates introduced above.

The correctness of a Gamma program can be established by showing that it satisfies a number of properties which together imply the postcondition. A good start for deducing properties of the output of a Gamma program is by calculating its termination condition. In addition to the termination condition, it may be necessary to find a suitable collection of invariants such that their conjunction implies the postcondition.

In the next section we will briefly illustrate this method by proving the correctness of the Gamma program *swap* for sorting from Example 2.1.1.

Correctness of the Sorting Program

We assume the input to the sorting program *swap* is a sequence $\overline{a_0} = \langle\, a_1, \ldots, a_n \,\rangle$. The sorting program is correct if it produces a rearrangement of the elements of the sequence in nondecreasing order.

Definition 2.2.2 *A sequence* $\bar{l} = \langle\, l_1, \ldots, l_n \,\rangle$ *is sorted iff*

$$\forall i, j : 1 \leq i, j \leq n : i < j \Rightarrow l_i \leq l_j \tag{2.2}$$

A pair of elements from the sequence which violates (2.2), is called an *inversion*. Hence, the sorted sequence is characterized by having no inversion.

The initial sequence $\overline{a}$ is represented by the multiset $M_0 = \{(a_i, i) \mid 1 \leq i \leq n\}$. The Gamma program for sorting consists of the single rewrite rule *swap* which exchanges two elements that form an inversion.

$$swap \mathrel{\widehat{=}} (i, x), (j, y) \mapsto (i, y), (j, x) \Leftarrow x > y \ \wedge \ i < j \tag{2.3}$$

Generally, the postcondition of a Gamma program falls into two parts: an existential part and a universal part . The existential part states that certain elements are present in the multiset and the universal part states that these elements are a solution to the problem.

To formally express the postcondition for the sorting program we introduce the following auxiliary definition.

Definition 2.2.3 *Let* $\bar{l}$ *be a sequence and let* k *be some value. Then* $\bar{l} \downarrow k$ *denotes the number of occurrences of* k *in* $\bar{l}$*.*

For an initial sequence $\overline{a}$, the postcondition can be expressed as follows

1. Existential: $\forall i : 1 \leq i \leq n : \#(i, x) = 1$
2. Universal:
   (a) $\forall i : 1 \leq i \leq n : \overline{a} \downarrow a_i = (\#(j, x) : x = a_i)$
   (b) $\forall i, j, x, y : (i, x), (j, y) : i < j \Rightarrow x \leq y$

We will proceed according to the following strategy. First, we will calculate the termination predicate of the sorting program. Next, we examine which properties need to hold in addition to the termination predicate such that the postcondition is met and attempt to prove one or more invariants which imply these additional properties. Finally, we prove that the program terminates.

Hence, for the sorting program, we first calculate the termination predicate. The termination predicate $\dagger swap$ implies condition 2(b) of the postcondition. Next, we show that the remaining properties 1 and 2(a) are invariants of the sorting program *swap*.

Lemma 2.2.4 *invariant* $\forall i : 1 \leq i \leq n : \overline{a} \downarrow a_i = (\#(j, x) : x = a_i)$

Proof

- *initially* : follows from the definition of M_0.
- *stable* : We show that the property is preserved by every possible individual execution of *swap*. Assume the property holds in M and $\langle swap, M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle swap, M'\rangle$. Hence $\sigma = \{(i, x), (j, y)\}/\{(j, x), (i, y)\}$ where $i < j$ and $x > y$. From the fact that σ inserts the same values x and y as it removes, follows that the property continues to hold.

□

Lemma 2.2.5 *invariant* $\forall i : 1 \leq i \leq n : \#(i, x) = 1$

Proof Analogous to Lemma 2.2.4. □

We finish by showing that the program *swap* terminates. To this end, we define a metric I that maps a multiset M onto the number of inversions in (the sequence represented by) M

$$I(M) = (+i, j, x, y : (i, x), (j, y) \in M : i > j \wedge x < y : 1)$$

Because the initial sequence is finite, the number of inversions is finite. Furthermore, $\langle swap, M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle swap, M'\rangle$ implies $I(M') < I(M)$. The number of inversions is bounded from below because there can be no fewer than zero inversions. We conclude that the program terminates.

2.3 Concluding Remarks

In this chapter we have presented the Gamma model and provided it with a formal semantics in terms of a labelled multi-step transition system. This semantics will be used in the next chapter for defining the semantics of a coordination language for Gamma.

Based on the semantics of Gamma we have defined a logic for reasoning about programs. We illustrated a method for using this logic by showing how it can be used to prove the correctness of a sorting program. A more elaborate example of the application of the logic can be found in Chapter 7 where we use it to prove the correctness of a Gamma program for solving triangular systems of linear equations.

3 The Coordination Model

In Chapter 2 we presented the Gamma programming model which allows the basic computations of a program to be expressed in a concise way and with a minimum of control. This enables the programmer to define the functional aspects of a program while deferring behaviour related decisions until a second stage in the design process. In support of this second activity we next introduce a coordination language that exploits the highly nondeterministic behaviour of Gamma to impose additional control with the objective to improve efficiency.

3.1 The Coordination Language

We refer to programs that are written in the coordination language as *schedules* to emphasize the fact that they are not really programs but rather execution plans or harnesses for an existing Gamma program. A schedule is an expression that represents an imperative statement over the rules from a Gamma program. The basic construction for schedules (next to skip which denotes the empty schedule) is the rule-conditional $r \rightarrow s[t]$.

Here r is a multiset rewrite rule and s and t denote arbitrary schedules. This schedule is executed by first attempting to execute the rule r, if this succeeds, then execution continues with the schedule s. If execution of r fails, then execution continues with t. As a notational convention, we write $r \rightarrow s[\mathsf{skip}]$ as $r \rightarrow s$ and $r \rightarrow \mathsf{skip}$ as r.

The coordination language provides a number of combinators that can be used to build more complex schedules. The complete set of combinators that is included in the kernel language is defined by the following abstract syntax for schedules. We use $\mathbb{S}$ to denote the set of schedule expressions, ranged over by s, t, u. The set $\mathcal{S}$ denotes the set of schedule identifiers , ranged over by S, T. A schedule without free schedule variables is called a *ground schedule* . The set of ground schedules is denoted $\mathbb{S}_{ground}$. The substitution of schedule(s) $\overline{t}$ for variable(s) $\overline{X}$ in a schedule s is written $s\{\overline{t}/\overline{X}\}$. A

sequence of values is denoted by $\overline{v}$. Variables that range over these values are denoted by $\overline{x}, \overline{y}$.

Syntactic Categories

c	$\in$	*Boolean Expression*
r	$\in$	*Rule*
s	$\in$	*Schedule Expression*
S	$\in$	*Schedule Identifier*

Definition

s	::=	skip
		$r \rightarrow s[s]$
		$s \,;\, s$
		$s \parallel s$
		$c \triangleright s[s]$
		$!s$
		$S(\overline{v})$ where $S(\overline{x}) \mathrel{\widehat{=}} s$

Figure 3.1: Abstract Syntax of the Coordination Language

Schedules can be composed sequentially, using the combinator ";" and be composed in parallel using " $\parallel$ " . The execution of a parallel composition $s \parallel t$ proceeds by a step performed by either s or t, or by a parallel step in which both s and t participate. For notational convenience, we write s^k, for $k \geq 0$, to denote k copies of schedule s composed in parallel. Formally, $s^0 = \mathsf{skip}$ and for $k > 0$, $s^k = s \parallel s^{k-1}$. Furthermore, we use $\Pi_{i=1}^{n} s_i$ to denote $s_1 \parallel s_2 \parallel \ldots \parallel s_n$.

Execution of a Gamma program is such that the number of rules that may be executed varies dynamically with the number of available elements in the multiset. In order to describe this dynamic behaviour using schedules, the replication operator "!" is included. The schedule $!s$ denotes an arbitrary number of copies of s executing in parallel.

The occurrence of a schedule identifier $S(\overline{v})$ is accompanied by a corresponding schedule definition of the form $S(\overline{x}) \mathrel{\widehat{=}} s$. The free variables in s are taken from the sequence $\overline{x}$. Schedule definitions are included for structuring purposes, as well as a means to express recursive schedules. The use of recursion is typically accompanied by the use of a conditional schedule $c \triangleright s[t]$ where c represents a boolean expression. If c evaluates to *true*, then schedule s is executed, otherwise execution continues with t. Analogously to the rule-conditional, $c \triangleright s[\mathsf{skip}]$ is written as $c \triangleright s$. The variables used in conditions c, typically taken from $\overline{x}$, are called *control variables* . Note that the truth value of c does

not depend on the state of the multiset.

In Gamma, nondeterminism arises at two levels:

1. at the selection of a rewrite-rule,

2. in selecting elements from the multiset.

The coordination language as introduced so far is only capable of resolving the first type of nondeterminism. The second type is resolved by strengthening (or specializing) the reaction condition of a rewrite-rule.

Definition 3.1.1

1. *A rewrite rule* $r' = \overline{x}' \mapsto m' \Leftarrow b'$ *is a* strengthening *of a rewrite rule* $r = \overline{x} \mapsto m \Leftarrow b$, *denoted* $r' \lhd r$, *if* $\overline{x} = \overline{x}'$, $\overline{m} = \overline{m'}$ *and* $b' \Rightarrow b$.

2. *If* R' *and* R *are sets of rewrite rules, then we write* $R' \lhd R$ *if* $\forall r' \in R' : (\exists r \in R : r' \lhd r)$.

Rather than scheduling a rewrite rule r directly, we can schedule a rewrite rule r', such that $r' \lhd r$. Because the reaction condition of r' is a strengthening of that of r, there are fewer (combinations of) elements from the multiset that satisfy this condition. Hence, rule r' exhibits restricted behaviour compared to r.

To illustrate, we return to the sorting program from Example 2.1.1 which consists of the rewrite rule *swap*. A schedule that, for instance, exchanges neighbouring values only, will make use of a rule *swap*′ which is obtained from the original rule by strengthening condition $i < j$ to $i = j - 1$ to get

$$swap' \mathrel{\widehat{=}} (i, x), (j, y) \mapsto (i, y), (j, x) \Leftarrow i = j - 1 \ \wedge \ x > y$$

To facilitate this process we shall adopt the notational convention that definitions of rewrite rules may be parameterized by variables that are used to narrow the set of eligible elements from the multiset. For the sorting program, we can define the following (family of) rule(s)

$$swap(i, j) \mathrel{\widehat{=}} (i, x), (j, y) \mapsto (i, y), (j, x) \Leftarrow i < j \ \wedge x > y$$

We can now specify a coordination strategy that schedules the sorting program *swap* such that it behaves like, for instance, insertion sort as $\mathit{InsertionSort}(1)$ where

$$\begin{array}{lcl} \mathit{InsertionSort}(i) & \widehat{=} & (i \leq n) \triangleright (\mathit{Insert}(i); \mathit{InsertionSort}(i+1)) \\ \mathit{Insert}(i) & \widehat{=} & (i > 0) \triangleright (\mathit{swap}(i-1, i) \rightarrow \mathit{Insert}(i-1)) \end{array}$$

Here n denotes the length of the sequence. A well known parallel sorting algorithm (see e.g. [102]) is Odd-Even Transposition Sort. The coordination strategy $\mathit{OddEvenSort}(n)$ (defined below) imposes an ordering on the execution of the sorting program *swap* such that it corresponds to the Odd-Even Transposition Sort algorithm.

$$\begin{array}{lcl} \mathit{OddEvenSort}(m) & \widehat{=} & (m \geq 0) \triangleright (\mathit{Odd}\ ;\ \mathit{Even}\ ;\ \mathit{OddEvenSort}(m-2)) \\ \mathit{Odd} & \widehat{=} & \Pi_{i=0}^{n \,\mathtt{div}\, 2-1} \mathit{swap}(2i+1, 2i+2) \\ \mathit{Even} & \widehat{=} & \Pi_{i=0}^{n+1 \,\mathtt{div}\, 2-1} \mathit{swap}(2i, 2i+1) \end{array}$$

Both of the above schedules describe a particular method of executing the sorting program *swap*. However, it has not been shown that these schedules actually steer the Gamma program such that it yields the required result.

Limiting the rules that are used in a schedule to those rules that appear in a given Gamma program, ensures that the schedule does not define behaviour that can not be matched by that Gamma program. However, a schedule may be at fault if it terminates before a final state of the Gamma program has been reached. In that case, the schedule only describes a prefix of a computation of the Gamma program.

Because schedules can be quite complicated, it is desirable to use a rigorous method for reasoning about their correctness. In the next section, we present a formal semantics for the coordination language which may serve as the basis for such methods of reasoning.

3.2 Semantics of the Coordination Language

The operational semantics of the coordination language is defined in Figure 3.3 as a labelled multi-step transition relation between configurations $\langle s, M \rangle$ where s is a schedule and M is a multiset. The labels λ of transitions either denote a multiset substitution or the special symbol ε which indicates transitions that do not affect the multiset. The ε symbol is a left and right unit for composition of multiset substitutions $(\cdot)$; i.e. $\varepsilon \cdot \lambda = \lambda = \lambda \cdot \varepsilon$.

The semantics of each combinator from the coordination language is defined by one

or more inference rules. The semantics of the schedule language is linked to that of the Gamma programs through the inference rules $(N0)$ and $(N1)$ for the rule conditional combinator. This construction enables the coordination language to schedule the individual computations as they are defined by the rules of a given Gamma program.

The set of semantic rules has been kept concise by identifying expressions that are structurally equivalent. A typical case is commutativity of parallel composition: "$s_1 \parallel s_2$" and "$s_2 \parallel s_1$" are equivalent ways of writing down the same schedule. The ordering in which the composition is written, should not make a difference. We therefore define a *structural congruence* "$\equiv$" to be the smallest congruence relation over a set of terms such that a number of laws hold. Terms are thus grouped together on the basis of their syntax, allowing the semantic rules to focus on behavioural aspects of the terms. This method of separating structural from behavioural issues was inspired by work on the chemical abstract machine [15].

The structural congruences used by the operational semantics for schedules are given in Figure 3.2. Note that the use of the structural congruences (E6), (E7) and (E9) omit the need for explicit semantic rules for the conditional $c \triangleright s[t]$ and recursion.

$$
\begin{array}{ll}
(E1) & \mathsf{skip}; s \equiv s \\
(E2) & s_1; (s_2; s_3) \equiv (s_1; s_2); s_3 \\
(E3) & \mathsf{skip} \parallel s \equiv s \\
(E4) & s_1 \parallel s_2 \equiv s_2 \parallel s_1 \\
(E5) & s_1 \parallel (s_2 \parallel s_3) \equiv (s_1 \parallel s_2) \parallel s_3 \\
(E6) & \mathit{true} \triangleright s[t] \equiv s \\
(E7) & \mathit{false} \triangleright s[t] \equiv t \\
(E8) & !\mathsf{skip} \equiv \mathsf{skip} \\
(E9) & S(\overline{v}) \equiv s[\overline{x} := \overline{v}] \qquad \text{if } S(\overline{x}) \mathrel{\widehat{=}} s
\end{array}
$$

Figure 3.2: Structural Congruences for Schedules

$$(N0) \qquad \frac{\langle r, M\rangle\surd}{\langle r \to s[t], M\rangle \xrightarrow{\varepsilon} \langle t, M\rangle}$$

$$(N1) \qquad \frac{\langle r, M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle r, M'\rangle}{\langle r \to s[t], M\rangle \xrightarrow{\sigma} \langle s, M'\rangle}$$

$$(N2) \qquad \frac{\langle s_1, M\rangle \xrightarrow{\lambda} \langle s_1', M'\rangle}{\langle s_1 \parallel s_2, M\rangle \xrightarrow{\lambda} \langle s_1' \parallel s_2, M'\rangle}$$

$$(N3) \qquad \frac{\langle s_1, M\rangle \xrightarrow{\lambda} \langle s_1', M'\rangle \quad \langle s_2, M\rangle \xrightarrow{\varepsilon} \langle s_2', M\rangle}{\langle s_1 \parallel s_2, M\rangle \xrightarrow{\lambda} \langle s_1' \parallel s_2', M'\rangle}$$

$$(N4) \qquad \frac{\langle s_1, M\rangle \xrightarrow{\sigma_1} \langle s_1', M_1\rangle \quad \langle s_2, M\rangle \xrightarrow{\sigma_2} \langle s_2', M_2\rangle}{\langle s_1 \parallel s_2, M\rangle \xrightarrow{\sigma_1\cdot\sigma_2} \langle s_1' \parallel s_2', M[\sigma_1 \cdot \sigma_2]\rangle} \quad \text{if } M \models \sigma_1 \bowtie \sigma_2$$

$$(N5) \qquad \frac{\langle s_1, M\rangle \xrightarrow{\lambda} \langle s_1', M'\rangle}{\langle s_1; s_2, M\rangle \xrightarrow{\lambda} \langle s_1'; s_2, M'\rangle}$$

$$(N6) \qquad \frac{\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle}{\langle !s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle}$$

$$(N7) \qquad \frac{\langle s \parallel !s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle}{\langle !s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle}$$

$$(N8) \qquad \frac{t \equiv s \quad \langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle \quad s' \equiv t'}{\langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle}$$

Figure 3.3: Semantics of Schedules

Analogous to the reflexive transitive transition relation for programs, we define the reflexive transitive closure of the transition relation for schedules.

Definition 3.2.1

$$\langle s, M\rangle \xrightarrow{\langle\,\rangle}{}^{*} \langle s, M\rangle \qquad \frac{\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle}{\langle s, M\rangle \xrightarrow{\lambda}{}^{*} \langle s', M'\rangle} \qquad \frac{\langle s, M\rangle \xrightarrow{\overline{\lambda_1}}{}^{*} \langle s', M'\rangle \quad \langle s', M'\rangle \xrightarrow{\overline{\lambda_2}}{}^{*} \langle s'', M''\rangle}{\langle s, M\rangle \xrightarrow{\overline{\lambda_1}\cdot\overline{\lambda_2}}{}^{*} \langle s'', M''\rangle}$$

The reflexive transitive transition relation uses labels $\overline{\lambda}$ which denote sequences of individual labels. For convenience we identify the singleton sequence $\langle\,\lambda\,\rangle$ with its only element λ. Furthermore, we use $\widehat{\lambda}$ to denote the sequence $\overline{\lambda}$ where all occurrences of ε have been removed.

Analogous to the case for Gamma programs, we define a capability function for schedules which models their input-output behaviour. The capability of a configuration is defined as the set of possible multisets it may produce, plus the special symbol $\perp$ if the configuration may never terminate.

Definition 3.2.2 *We define the "may diverge" predicate $\uparrow$ on configurations:*
$\langle s, M\rangle\uparrow$ *if and only if*
$\langle s, M\rangle = \langle s_0, M_0\rangle$ *and for all $i \geq 0$ there exists a λ_i such that $\langle s_i, M_i\rangle \xrightarrow{\lambda_i} \langle s_{i+1}, M_{i+1}\rangle$*

Definition 3.2.3 *The capability function $\mathcal{C} : \mathbb{S} \times \mathbb{M} \to \mathcal{P}(\mathbb{M}) \cup \{\perp\}$ for schedules, is defined as*

$$\mathcal{C}(s, M) = \{\perp \mid \langle s, M\rangle\uparrow\} \cup \{M' \mid \langle s, M\rangle \xrightarrow{\overline{\lambda}}{}^{*} \langle \mathsf{skip}, M'\rangle\}$$

3.2.1 Rationale for the Coordination Language

The behaviour of Gamma programs ranges from highly nondeterministic chaotic execution to that of known algorithms. We want to be able to express all the possible behaviours of Gamma programs in the same formalism. To this end we designed the coordination language. The combinators that are present in the coordination language have been chosen for one of two reasons.

- A combinator is needed for describing an ordering on the execution of actions.

- For all practical purposes, we need the schedule-representation of any of these orderings of actions to be finite. Some aspects of coordination strategies, like the number of iterations and the number of actions that can be executed in parallel, cannot in general be defined a priori. Hence we need constructs that evolve dynamically as a function of the input (rather than of the size of the input only).

In order to define an ordering on actions we need to describe two things:

- the precedes/succeeds relations between actions. This is traditionally represented by the ';' symbol: '$s_1 \,;\, s_2$' means that before the actions of 's_2' may be executed, all actions of 's_1' must be finished.

- the fact that actions are unordered. In our setting of schedules, the unorderedness of actions means that they can be executed concurrently. We write '$s_1 \parallel s_2$' to indicate that independent actions of 's_1' and 's_2' may be executed concurrently.

Finite representations of potentially infinite schedules can only be obtained by operators that evolve dynamically:

- Generally, the exact execution ordering of individual rules cannot be known in advance. Recursion is incorporated to describe iterations of arbitrary length. The unfolding of a recursive schedule typically depends on the given multiset. Choices based on the parameters of a schedule can be specified using the conditional construct '$c \triangleright s[t]$'.

- We do not know in advance how many rules may be executed concurrently at any stage in the computation. The schedule '$!s$' may evolve dynamically into the number of copies of 's' that is needed. Hence, replication describes an arbitrary degree of parallelism.

We briefly reflect on the differences between the way we use replication and the way it is used by Milner in his π-calculus [91].

In the π-calculus replication is used to simulate recursion and is therefore chosen, in place of recursion, as a primitive notion. Milner uses replication to describe the possibility of spawning an infinite number of copies of a process. However, (the observation of) the spawning of processes is triggered by communicating with other processes that are executing in parallel. Hence, the actual number of copies that is spawned can be determined by the environment.

In our setting, the replication of schedules (which correspond to Milner's processes) is autonomous: the number of times a schedule is replicated can not be influenced by other schedules that are running in its context.

It is desirable that replication stops when the schedule/process that is spawned no longer contributes to the outcome of the computation. In the π-calculus, the environment has control over the spawning of processes, hence the environment can decide when replication may stop. However, in our coordination language replication is autonomous, hence the ability to stop has to be built into the semantics of replication. This is achieved by the inclusion of the semantic rule (N6).

If we would use recursion to define this potentially finite behaviour, we would also need a combinator for (pre-emptive) nondeterministic choice. In Section 9.2.1 we describe this construction and explain why we do not want the combinator for nondeterministic choice in our kernel language.

3.2.2 Single-Step Transitions

The operational semantics of Figure 3.3 describes behavioural aspects of our coordination language. A particular aspect of interest is the parallelism in the behaviour of coordination strategies. The transition system that we use to define the operational semantics of schedules is a *multi-step* transition system. Characteristic of multi-step transition systems is that multiple actions, in our case: multiset rewrites, may be captured by a single transition, thereby modelling the possibility of parallel execution. This contrasts to *single-step* transition systems, such as used in [66] to define the semantics of Gamma programs, where every individual transition corresponds to precisely one rewrite. An important feature of multi-step transition systems is that they distinguish parallel execution from interleaved execution which the single-step transition systems do not. In Section 9.2.3, we show that, due to this property, multi-step transition systems more adequately model parallel computation than single-step transition systems.

The choice in favour of a multi-step transition system brings a difficulty along with it. While the multi-step transition approach is favourable for modelling behaviour, it is more difficult to reason about multi-step transitions than single-step transitions. The units of behaviour over which we will reason are individual transitions. In the case of a multi-step transition, we need to consider all possible combinations of multiset rewrites out of which it may be composed. This incurs a combinatorial explosion which complicates reasoning about multi-step transitions.

The one-to-one correspondence between transitions and rewrites, which holds for single-step transition systems, does not suffer from this combinatorial explosion, which makes it easier to reason about individual transitions.

To reduce the complexity of reasoning about multi-step transition, we next present a result which shows that every multi-step transition can be mimicked by a sequence of single-step transitions. This greatly facilitates reasoning about the behaviour of schedules because it can be used to reduce reasoning about parallel behaviour to reasoning about sequential behaviour.

The multi-step character of the semantics of the coordination language is due to inference rules $(N3)$ and $(N4)$. These inference rules cater for the derivation of transitions which model multiple concurrent rewrites. This observation allows single-step transitions to be characterized by their derivation tree.

Definition 3.2.4 *If a transition* $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$ *is derived without using the semantics rules* $(N3)$ *and* $(N4)$*, it is called a* single-step *transition.*

A property of the operational semantics in Figure 3.3 is that every multi-step transition can be split into a sequence of single-step transitions which has the same effect on the multiset. This has as a consequence that sequential behaviour is a special case of parallel behaviour.

Lemma 3.2.5 *If* $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$*, then there exists a sequence of single-step transitions*

$$\langle t_0, M_0\rangle \xrightarrow{\lambda_1} \langle t_1, M_1\rangle \ldots \xrightarrow{\lambda_i} \ldots \langle t_{n-1}, M_{n-1}\rangle \xrightarrow{\lambda_n} \langle t_n, M_n\rangle$$

such that $\langle s, M\rangle = \langle t_0, M_0\rangle$ *and* $\langle t_n, M_n\rangle = \langle s', M'\rangle$ *and* $\lambda = \lambda_1 \cdot \ldots \cdot \lambda_n$*.*

Proof By transition induction: Assume that $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$ is derived by some inference. We consider the the different ways in which the last step of this inference can be done.

- By (N0) or (N1). The transition is single-step by definition.
- By (N2), with $s \equiv s_1 \parallel s_2$, from $\langle s_1, M\rangle \xrightarrow{\lambda} \langle s_1', M'\rangle$. Hence $s' \equiv s_1' \parallel s_2$. Then by the induction hypothesis there exists a sequence of single-step transitions

$$\langle t_0, M_0\rangle \xrightarrow{\lambda_1} \langle t_1, M_1\rangle \ldots \xrightarrow{\lambda_i} \ldots \langle t_{n-1}, M_{n-1}\rangle \xrightarrow{\lambda_n} \langle t_n, M_n\rangle$$

where $\langle s_1, M\rangle = \langle t_0, M_0\rangle$, $\langle t_n, M_n\rangle = \langle s_1', M'\rangle$ and $\lambda = \lambda_1 \cdot \ldots \cdot \lambda_n$.
By repeated use of $(N2)$ we derive the following sequence of single-step transitions

$$\langle t_0 \parallel s_2, M\rangle \xrightarrow{\lambda_1} \langle t_1 \parallel s_2, M_1\rangle \ldots \xrightarrow{\lambda_i} \ldots \langle t_{n-1} \parallel s_2, M_{n-1}\rangle \xrightarrow{\lambda_n} \langle t_n \parallel s_2, M'\rangle$$

- By (N3), with $s \equiv s_1 \parallel s_2$, from $\langle s_1, M\rangle \xrightarrow{\lambda} \langle s_1', M'\rangle$ and $\langle s_2, M\rangle \xrightarrow{\varepsilon} \langle s_2', M\rangle$.
Hence $s' \equiv s_1' \parallel s_2'$. The induction hypothesis applies to both of these transitions. This gives the following sequences of single-step transitions

$$\langle t_{1,0}, M_0\rangle \xrightarrow{\lambda_1} \langle t_{1,1}, M_1\rangle \ldots \xrightarrow{\lambda_i} \ldots \langle t_{1,n_1-1}, M_{n_1-1}\rangle \xrightarrow{\lambda_{n_1}} \langle t_{1,n_1}, M_{n_1}\rangle$$

where $\langle s_1, M\rangle = \langle t_{1,0}, M_0\rangle$, $\langle t_{1,n_1}, M_{n_1}\rangle = \langle s_1', M'\rangle$ and $\lambda = \lambda_1 \cdot \ldots \cdot \lambda_{n_1}$;

$$\langle t_{2,0}, M\rangle \xrightarrow{\varepsilon} \langle t_{2,1}, M\rangle \ldots \xrightarrow{\varepsilon} \ldots \langle t_{2,n_2-1}, M\rangle \xrightarrow{\varepsilon} \langle t_{2,n_2}, M\rangle$$

where $s_2 = t_{2,0}$, $t_{2,n_2} = s_2'$ and $\varepsilon \cdot \ldots \cdot \varepsilon = \varepsilon$.
By repeated use of $(N2)$ we derive the following sequences of single-step transitions

$$\begin{gathered}\langle t_{1,0} \parallel t_{2,0}, M\rangle \xrightarrow{\varepsilon} \langle t_{1,0} \parallel t_{2,1}, M\rangle \\ \ldots \xrightarrow{\varepsilon} \ldots \\ \langle t_{1,0} \parallel t_{2,n_2-1}, M\rangle \xrightarrow{\varepsilon} \langle t_{1,0} \parallel t_{2,n_2}, M\rangle\end{gathered}$$

and

$$\begin{gathered}\langle t_{1,0} \parallel t_{2,n_2}, M\rangle \xrightarrow{\lambda_1} \langle t_{1,1} \parallel t_{2,n_2}, M_1\rangle \\ \ldots \xrightarrow{\lambda_i} \ldots \\ \langle t_{1,n_1-1} \parallel t_{2,n_2}, M_{n_1-1}\rangle \xrightarrow{\lambda_{n_1}} \langle t_{1,n_1} \parallel t_{2,n_2}, M'\rangle\end{gathered}$$

The result follows by concatenating these sequences:

$$\langle t_{1,0} \parallel t_{2,0}, M\rangle \xrightarrow{\varepsilon} \ldots \xrightarrow{\varepsilon} \langle t_{1,0} \parallel t_{2,n_2}, M\rangle \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n_1}} \langle t_{1,n_1} \parallel t_{2,n_2}, M'\rangle$$

Clearly $\varepsilon \cdot \ldots \cdot \varepsilon \cdot \lambda_1 \cdot \ldots \cdot \lambda_{n_1} = \lambda$.

- By (N4), with $s \equiv s_1 \parallel s_2$, from $\langle s_1, M\rangle \xrightarrow{\sigma_1} \langle s_1', M_1\rangle$ and $\langle s_2, M\rangle \xrightarrow{\sigma_2} \langle s_2', M_2\rangle$ where $M \models \sigma_1 \bowtie \sigma_2$ and $\sigma = \sigma_1 \cdot \sigma_2$. Hence $s' \equiv s_1' \parallel s_2'$. From Lemma A.2.6 follows that these transitions may be applied in arbitrary interleaved order; for instance

$$\langle s_1, M\rangle \xrightarrow{\sigma_1} \langle s_1', M_1\rangle \quad \text{and} \quad \langle s_2, M_1\rangle \xrightarrow{\sigma_2} \langle s_2', M'\rangle$$

Applying the induction hypothesis to each of these gives the following sequences of single-step transitions

$$\langle s_1, M\rangle \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n_1}} \langle s_1', M_1\rangle \quad \text{and} \quad \langle s_2, M_1\rangle \xrightarrow{\lambda_1'} \ldots \xrightarrow{\lambda_{n_2}'} \langle s_2', M'\rangle$$

where $\lambda_1 \cdot \ldots \cdot \lambda_{n_1} = \sigma_1$ and $\lambda_1' \cdot \ldots \cdot \lambda_{n_2}' = \sigma_2$. By repeated use of $(N2)$ we derive the following sequences of single-step transitions

$$\langle s_1 \parallel s_2, M\rangle \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n_1}} \langle s_1' \parallel s_2, M_1\rangle \quad \text{and} \quad \langle s_1' \parallel s_2, M_1\rangle \xrightarrow{\lambda_1'} \ldots \xrightarrow{\lambda_{n_2}'} \langle s_1' \parallel s_2', M'\rangle$$

We concatenate these sequences into

$$\langle s_1 \parallel s_2, M\rangle \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n_1}} \langle s_1' \parallel s_2, M_1\rangle \xrightarrow{\lambda_1'} \ldots \xrightarrow{\lambda_{n_2}'} \langle s_1' \parallel s_2', M'\rangle$$

And $\lambda_1 \cdot \ldots \cdot \lambda_{n_1} \cdot \lambda_1' \cdot \ldots \cdot \lambda_{n_2}' = \sigma_1 \cdot \sigma_2 = \sigma$.

- By (N5) with $s \equiv s_1; s_2$. The proof is analogous to the case for (N2).

- By (N6), with $s \equiv !s$, from $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$. From the induction hypothesis follows that there exists a sequence

$$\langle s_0, M_0\rangle \xrightarrow{\lambda_1} \langle s_1, M_1\rangle \ldots \xrightarrow{\lambda_i} \ldots \langle s_{n-1}, M_{n-1}\rangle \xrightarrow{\lambda_n} \langle s_n, M_n\rangle$$

of single-step transitions such that $\langle s, M\rangle = \langle s_0, M_0\rangle$, $\langle s_n, M_n\rangle = \langle s', M'\rangle$ and $\lambda_1 \cdot \ldots \cdot \lambda_n = \lambda$. For the first transition we use $(N6)$ to derive $\langle !s, M\rangle \xrightarrow{\lambda_1} \langle s_1, M_1\rangle$ (which is single-step). Concatenation gives $\langle !s, M\rangle \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_n} \langle s', M'\rangle$.

- By (N7), with $s \equiv !s$, from $\langle s \parallel !s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$. From the induction hypothesis follows that there exists a sequence

$$\langle s_0, M_0\rangle \xrightarrow{\lambda_1} \langle s_1, M_1\rangle \ldots \xrightarrow{\lambda_i} \ldots \langle s_{n-1}, M_{n-1}\rangle \xrightarrow{\lambda_n} \langle s_n, M_n\rangle$$

of single-step transitions where $\langle s \parallel !s, M\rangle = \langle s_0, M_0\rangle$, $\langle s_n, M_n\rangle = \langle s', M'\rangle$ and $\lambda_1 \cdot \ldots \cdot \lambda_n = \lambda$. For the first transition we use $(N7)$ to infer $\langle !s, M\rangle \xrightarrow{\lambda_1} \langle s_1, M_1\rangle$ (which is single-step). Concatenation with the subsequent transitions gives $\langle !s, M\rangle \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_n} \langle s', M'\rangle$.

- By (N8), from $s \equiv t$, $s' \equiv t'$ and $\langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle$. By the induction hypothesis

there exists a sequence of single-step transitions

$$\langle t_0, M_0\rangle \xrightarrow{\lambda_1} \langle t_1, M_1\rangle \ldots \xrightarrow{\lambda_i} \ldots \langle t_{n-1}, M_{n-1}\rangle \xrightarrow{\lambda_n} \langle t_n, M_n\rangle$$

where $\langle t_0, M_0\rangle = \langle t, M\rangle$ and $\langle t_n, M_n\rangle = \langle t', M'\rangle$ and $\lambda_1 \cdot \ldots \cdot \lambda_n = \lambda$. From the first and the last transitions of this sequence we infer, by (N8), $\langle s, M\rangle \xrightarrow{\lambda_1} \langle t_1, M_1\rangle$ and $\langle t_{n-1}, M_{n-1}\rangle \xrightarrow{\lambda_n} \langle s', M'\rangle$ (which are both single-step). Concatenation gives

$$\langle s, M\rangle \xrightarrow{\lambda_1} \langle t_1, M_1\rangle \ldots \xrightarrow{\lambda_i} \ldots \langle t_{n-1}, M_{n-1}\rangle \xrightarrow{\lambda_n} \langle s', M'\rangle$$

□

3.3 Most General Schedules

The coordination language allows us to specify behaviours from a wide spectrum of possibilities, ranging from the completely deterministic behaviour of known algorithms to the chaotic execution of Gamma programs. The latter can be seen by constructing a schedule that comprises all possible behaviours of a Gamma program. We refer to this schedule as the *most general schedule* . The most general schedule can be defined compositionally on the structure of Gamma programs (as given by the abstract syntax of Figure 2.2 in Chapter 2).

Definition 3.3.1 *Let R denote a simple program $r_1 + r_2 + \cdots r_n$ and let P_1 and P_2 be two arbitrary Gamma programs. The most general schedules for R and $P_1 \circ P_2$ are defined by*

$$\begin{array}{lcl} \Gamma_R & \widehat{=} & !\,(r_1 \to \Gamma_R \parallel r_2 \to \Gamma_R \parallel \cdots \parallel r_n \to \Gamma_R) \\ \Gamma_{P_1 \circ P_2} & \widehat{=} & \Gamma_{P_2}; \Gamma_{P_1} \end{array}$$

First, we give an informal explanation of the construction of the most general schedule. In the remainder of this chapter we formally prove the equivalence between Gamma programs and their most general schedule.

All rules r_i of a simple program R are composed in parallel in Γ_R such that initially any (combination of) rule(s) may be executed. The replication that occurs in Γ_R allows the schedule to exhibit an arbitrary degree of concurrency (like the corresponding chaotic program). Hence, for the most general schedule, any independent number of instances of

the constituent rules may be executed in parallel. Successful execution of a rewrite rule may enable another rule, or re-enable itself. In order to avoid premature termination of the most general schedule, it is necessary that after the successful execution of a rule, every rule is tried (again) for execution. This is achieved by the recursive invocation of Γ_R by every rule-conditional. The definition of $\Gamma_{P_1 \circ P_2}$ is straightforward.

Whereas the behaviour of Gamma programs is implicit in their representation, the most general schedule explicitly represents this behaviour. This explicit representation makes it amenable to formal manipulation. The particular kind of manipulation that we are interested in is refinement of behaviour. The fact that a most general schedule describes all possible behaviours of a corresponding Gamma program allows it to be used as the starting point in a process of refinement aimed at deriving more specific execution strategies. The techniques necessary for refinement will be developed in Chapter 4.

3.3.1 Completeness of the Most General Schedule

Most general schedules play an important rôle in the process of refinement. They provide the initial description of the behaviour of Gamma programs. The central theme of this section is to show that the most general schedule deserves its name. To this end, we show that the most general schedule satisfies the following properties:

- Firstly, the most general schedule describes all possible ways of executing a Gamma program, but no more.
- Secondly, the input-output relation of a most general schedule matches that of the corresponding Gamma program.

In order to prove the above properties, we will introduce some auxiliary results and notation related to most general schedules.

Definition 3.3.2 *Let $P = r_1 + \cdots + r_n$ and let Γ_P be its most general schedule.*

$$\begin{aligned} \Pi_P &\equiv (r_1 \to \Gamma_P \,\|\, \cdots \,\|\, r_n \to \Gamma_P) \\ \Delta_{P,i} &\equiv (r_1 \to \Gamma_P \,\|\, \cdots \,\|\, r_{i-1} \to \Gamma_P \,\|\, r_{i+1} \to \Gamma_P \,\|\, \cdots \,\|\, r_n \to \Gamma_P) \end{aligned}$$

A term $\Delta_{P,i}$ differs from Π_P because it misses the i^{th} term $r_i \to \Gamma_P$. From commutativity and associativity of parallel composition ($\|$) follows that $\Pi_P \equiv (r_i \to \Gamma_P) \,\|\, \Delta_{P,i}$. Note that for simple programs P the most general schedule Γ_P can be written $!\Pi_P$.

We introduce the notion of *derivedness*. This notion relates a configuration to the configurations that it may evolve into by execution.

Definition 3.3.3

1. *We say that a configuration* $\langle s', M' \rangle$ *is* $\langle s, M \rangle$*-derived if* $\langle s, M \rangle \xrightarrow{\overline{\lambda}}{}^{*} \langle s', M' \rangle$ *for some* $\overline{\lambda}$*. This is also denoted* $\langle s, M \rangle \longrightarrow^{*} \langle s', M' \rangle$*.*

2. *We say that a schedule* s' *is* s*-derived if* $\langle s, M \rangle \longrightarrow^{*} \langle s', M' \rangle$ *for some* M *and* M'*.*

The predicate μ is used to denote a class of schedules which satisfy a certain syntactical format whose importance will be illustrated shortly by Lemma 3.3.7.

Definition 3.3.4 *Let* $S \mathrel{\widehat{=}} !(r_1 \to S \parallel \ldots \parallel r_n \to S)$*.*

1. *We write* $\mu_S(s)$ *if* $s \equiv (r_1 \to S)^{a_1} \parallel \ldots \parallel (r_n \to S)^{a_n} \parallel S^k$ *with* $a_i \geq 0$ *for all* $i : 1 \leq i \leq n$ *and* $k \geq 0$*.*

2. *We write* $\mu_S^{+}(s)$ *if* $\mu_S(s)$ *with* $k \geq 1$*; in other words* $s \equiv s' \parallel S$ *where* $\mu_S(s')$*.*

Next, we extend the use of predicate μ to configurations.

Definition 3.3.5 *Let* $S \mathrel{\widehat{=}} !(r_1 \to S \parallel \ldots \parallel r_n \to S)$*. We write* $\mu_S(s, M)$*, if the following conditions hold for* $\langle s, M \rangle$*:*

1. $s \equiv (r_1 \to S)^{a_1} \parallel \ldots \parallel (r_n \to S)^{a_n} \parallel S^k$ *with* $a_i \geq 0$ *for all* $i : 1 \leq i \leq n$ *and* $k \geq 0$

2. $(k = 0) \Rightarrow (\forall i : 1 \leq i \leq n : a_i = 0 \;\Rightarrow\; [\![\dagger r_i]\!] M)$

Next, we show that μ_{Γ_P} describes a relation between schedule and multiset that holds for any $\langle \Gamma_P, M \rangle$-derived configuration (for some simple program P). To this end, we first observe that the configuration $\langle \Gamma_P, M \rangle$ satisfies μ_{Γ_P} (for any M and simple P). Secondly we prove that property μ_S (for configurations) is invariant with respect to sequences of multi-step transitions.

The proof of invariance of μ_S with respect to sequences of multi-step transitions is structured as follows: First we show invariance of μ_S with respect to single-step transitions (Lemma 3.3.6). Using this, we show invariance for multi-step transitions (Lemma 3.3.7) and subsequently, we generalize the previous result to sequences of multi-step transitions (Lemma 3.3.8).

Lemma 3.3.6 *If* $\mu_S(s, M)$ *and* $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$ *is a single-step transition, then* $\mu_S(s', M')$.

Proof From $\mu_S(s, M)$ follows $s \equiv (r_1 \to S)^{a_1} \parallel \ldots \parallel (r_n \to S)^{a_n} \parallel S^k$ with $a_i \geq 0$ for all $i : 1 \leq i \leq n$ and $k \geq 0$ such that $(k = 0) \Rightarrow (\forall i : 1 \leq i \leq n : a_i = 0 \;\Rightarrow\; [\![\dagger r_i]\!]M)$.

We show $\mu_S(s', M')$ by induction on k.

- $k = 0$: By transition induction can be shown that a single-step transition can be derived in one of the following ways.
  - By (N0) from $\langle r_i, M\rangle\surd$ for some i. Hence $\lambda = \varepsilon$ and $M' = M$. Then $a'_j = a_j$ for all $j \neq i$, $a'_i = a_i - 1$ and $k' = k$. Hence $\mu_S(s', M')$.
  - By (N1) from $\langle r_i \to S, M\rangle \xrightarrow{\sigma} \langle S, M'\rangle$, for some i. Hence $\lambda = \sigma$. Then $a'_j = a_j$ for all $j \neq i$, $a'_i = a_i - 1$ and $k' = k + 1$. Hence $\mu_S(s', M')$.
- $k > 0$: By transition induction can be shown that a single step transition can be derived in one of the following ways.
  - By (N0) or (N1). The proof proceeds analogously to the case $k = 0$.
  - By (N8) from unfolding the definition of S. A transition can be derived from $\langle s'', M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$ where $s'' \equiv (r_1 \to S)^{a''_1} \parallel \ldots \parallel (r_n \to S)^{a''_n} \parallel S^{k''}$ with $a''_j = a_j + 1$ for all j and $k'' = k - 1$. Then $\mu_S(s'', M)$, hence the result follows by the induction hypothesis.

□

Now, we use the invariance of μ_S over single-step transitions to prove the invariance over multi-step transitions.

Lemma 3.3.7 *If* $\mu_S(s, M)$ *and* $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$, *then* $\mu_S(s', M')$.

Proof By Lemma 3.2.5 follows that there exist $\lambda_1, \ldots, \lambda_n, n \geq 1$ such that

$$\langle s_0, M_0\rangle \xrightarrow{\lambda_1} \langle s_1, M_1\rangle \ldots \xrightarrow{\lambda_i} \ldots \langle s_{n-1}, M_{n-1}\rangle \xrightarrow{\lambda_n} \langle s_n, M_n\rangle$$

where $\langle s, M\rangle = \langle s_0, M_0\rangle$ and $\langle s_n, M_n\rangle = \langle s', M'\rangle$ and each transition $\langle s_{i-1}, M_{i-1}\rangle \xrightarrow{\lambda_i} \langle s_i, M_i\rangle$, $1 \leq i \leq n$, is single-step.

By Lemma 3.3.6 follows that every single-step transition in this sequence preserves μ_S. Then by induction on the length of the sequence of single-step transitions, follows that $\mu_S(s', M')$. □

Lemma 3.3.8 generalizes the invariance of μ_S to sequences of multi-step transitions.

Lemma 3.3.8 *If $\mu_S(s, M)$ and $\langle s, M\rangle \xrightarrow{\overline{\lambda}}{}^* \langle s', M'\rangle$, then $\mu_S(s', M')$.*

Proof By induction on the length of the transition sequence.

- $|\lambda| = 0$: Then $\langle s', M'\rangle = \langle s, M\rangle$.
- $|\lambda| > 0$: Then transition sequence can be written as

$$\langle s, M\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle s'', M''\rangle \xrightarrow{\lambda''}{}^* \langle s', M'\rangle$$

  Because the transition sequence from s to s'' is shorter than the initial sequence, the induction hypothesis yields $\mu_S(s'', M'')$. Then, for the last transition, Lemma 3.3.7 yields $\mu_S(s', M')$.

□

From the fact that a most general schedules (for a simple program) satisfies μ_S follows from Lemma 3.3.7 that any configuration that a most general schedule evolves into also satisfies μ_S.

Corollary 3.3.9 *Let $S \mathrel{\widehat{=}} !(r_1 \to S \parallel \ldots \parallel r_n \to S)$.*
If $\langle S, M\rangle \longrightarrow^* \langle s', M'\rangle$, then $\mu_S(s', M')$.

Proof Straightforward from Lemma 3.3.8. □

We continue by showing that the most general schedule describes all possible ways for executing a Gamma program. We show this by first proving that the most general schedule describes all possible first transitions that the corresponding Gamma program may make. Next, we generalize this to sequences of transitions.

Lemma 3.3.10 shows that whatever transition a (simple) Gamma program may perform, this transition is also possible for the corresponding most general schedule. Furthermore, the form that the most general schedule arrives at after this transition, contains the most general schedule as a subterm that may contribute to the next transition.

Hence the schedule arrived at after a successful transition of the most general schedule has the potential of behaving again as the most general schedule.

Lemma 3.3.10 *Let $P = r_1 + \ldots + r_n$ with $n \geq 1$ be a simple Gamma program.*
If $\langle P, M\rangle \overset{\sigma}{\rightsquigarrow} \langle P, M'\rangle$, then $\langle \Gamma_P, M\rangle \xrightarrow{\sigma} \langle s, M'\rangle$ such that $s \equiv s' \parallel \Gamma_P$ for some $s' \in \mathbb{S}$.

Proof By transition induction on $\langle P, M\rangle \overset{\sigma}{\rightsquigarrow} \langle P, M'\rangle$.
The last step of the proof may have been derived using either rule (C2) or rule (C4) in the following ways.

- By (C2), then by (C1) and (C3) follows $\langle r_i, M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle r_i, M'\rangle$ for some $i : 1 \leq i \leq n$. By (N1) we infer $\langle r_i \to \Gamma_P, M\rangle \xrightarrow{\sigma} \langle \Gamma_P, M'\rangle$. Then, since $(r_i \to \Gamma_P) \parallel \Delta_{P,i} = \Pi_P$ we get from (N2) that $\langle \Pi_P, M\rangle \xrightarrow{\sigma} \langle \Gamma_P \parallel \Delta_{P,i}, M'\rangle$. By (N6) and $!\Pi_P = \Gamma_P$ we derive the transition $\langle \Gamma_P, M\rangle \xrightarrow{\sigma} \langle \Gamma_P \parallel \Delta_{P,i}, M'\rangle$.

- By (C4) from
$$\langle P, M\rangle \overset{\sigma_1}{\rightsquigarrow}_1 \langle P, M_1\rangle \tag{3.1}$$
and
$$\langle P, M\rangle \overset{\sigma_2}{\rightsquigarrow} \langle P, M_2\rangle \tag{3.2}$$
where $\sigma = \sigma_1 \cdot \sigma_2$ and $M \models \sigma_1 \bowtie \sigma_2$. From (3.2) we get from the induction hypothesis that
$$\langle \Gamma_P, M\rangle \xrightarrow{\sigma_2} \langle s', M_2\rangle \tag{3.3}$$
where $s' \equiv \Gamma_P \parallel s''$ for some schedule s''. By a derivation analogous to the case (C2) we deduce from (3.1) that
$$\langle \Pi_P, M\rangle \xrightarrow{\sigma_1} \langle \Gamma_P \parallel \Delta_{P,i}, M_1\rangle \tag{3.4}$$
From $M \models \sigma_1 \bowtie \sigma_2$, (3.3) and (3.4) we get using (N4) that
$$\langle \Pi_P \parallel \Gamma_P, M\rangle \xrightarrow{\sigma} \langle \Gamma_P \parallel \Delta_{P,i} \parallel s', M'\rangle \tag{3.5}$$
Because $!\Pi_P \equiv \Gamma_P$ we conclude using (N7) and (N8) that
$$\langle \Gamma_P, M\rangle \xrightarrow{\sigma} \langle \Gamma_P \parallel \Delta_{P,i} \parallel s', M'\rangle \tag{3.6}$$

□

The next lemma generalizes Lemma 3.3.10 by showing that the most general schedule can mimic any sequence of actions that a simple Gamma program may make.

Lemma 3.3.11 *Let $P = r_1 + \ldots + r_n$ be a simple Gamma program.*
If $\langle P, M\rangle \overset{\overline{\lambda}}{\rightsquigarrow}{}^* \langle P, M'\rangle$, then $\langle \Gamma_P, M\rangle \xrightarrow{\overline{\lambda}}{}^* \langle s, M'\rangle$ such that $s \equiv s' \,\|\, \Gamma_P$ for some $s' \in \mathbb{S}$.

Proof By induction on the length of the transition sequence.

- $\overline{\lambda} = \langle\ \rangle$: By reflexivity of $\longrightarrow^*$ follows $\langle \Gamma_P, M\rangle \xrightarrow{\langle\ \rangle}{}^* \langle \Gamma_P, M\rangle$.
- $\overline{\lambda} = \sigma \cdot \overline{\lambda'}$: hence $\langle P, M\rangle \overset{\sigma}{\rightsquigarrow} \langle P, M''\rangle$ and $\langle P, M''\rangle \overset{\overline{\lambda'}}{\rightsquigarrow}{}^* \langle P, M'\rangle$. For the former we get by Lemma 3.3.10 that $\langle \Gamma_P, M\rangle \xrightarrow{\sigma} \langle s \,\|\, \Gamma_P, M''\rangle$ for some $s \in \mathbb{S}$. For the latter we get from the induction hypothesis $\langle \Gamma_P, M''\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle s' \,\|\, \Gamma_P, M'\rangle$ for some $s' \in \mathbb{S}$. By (N2) we can glue these together to get $\langle \Gamma_P, M\rangle \xrightarrow{\overline{\lambda}}{}^* \langle s \,\|\, s' \,\|\, \Gamma_P, M'\rangle$.

□

The preceding lemma's show that a program and its most general schedule may perform the same (sequences of) transitions. Next, we show that the final states of a program and its most general schedule coincide. To this end, we show that in any state where a simple Gamma program terminates, any Γ_P-derived configuration may terminate without changing the multiset.

Lemma 3.3.12 *Let $P = r_1 + \ldots + r_n$ be a simple program and let $\langle s, M\rangle$ with $s \not\equiv \mathsf{skip}$ be $\langle \Gamma_P, M_0\rangle$-derived, for some M_0. If $\langle P, M\rangle\surd$, then $\langle s, M\rangle \xrightarrow{\varepsilon} \langle \mathsf{skip}, M\rangle$.*

Proof From $\langle P, M\rangle\surd$ follows by (C5) that $\langle r_i, M\rangle\surd$ for all $i : 1 \leq i \leq n$. Hence, by (N0) follows $\langle r_i \to \Gamma_P, M\rangle \xrightarrow{\varepsilon} \langle \mathsf{skip}, M\rangle$ for all $i : 1 \leq i \leq n$. Then, by (N2) and (N6) follows $\langle \Gamma_P, M\rangle \xrightarrow{\varepsilon} \langle \mathsf{skip}, M\rangle$.
By Lemma 3.3.7 follows $s \equiv (r_1 \to \Gamma_P)^{a_1} \,\|\, \ldots \,\|\, (r_n \to \Gamma_P)^{a_n} \,\|\, {\Gamma_P}^k$. By (N2) follows $\langle s, M\rangle \xrightarrow{\varepsilon} \langle \mathsf{skip}, M\rangle$. □

Lemma 3.3.13 shows that if after some sequence of transitions, a simple program terminates in some state, then the most general schedule may also terminate in that state after a sequence of transitions that differs from that of the program only with respect to ε-transitions.

Lemma 3.3.13 *Let P be a simple Gamma program. If $\langle P, M\rangle \overset{\overline{\lambda}}{\rightsquigarrow}{}^* \langle P, M'\rangle$ where $\langle P, M'\rangle\surd$, then $\langle \Gamma_P, M\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle \mathsf{skip}, M'\rangle$ such that $\widehat{\lambda'} = \overline{\lambda}$.*

Proof From $\langle P, M\rangle \overset{\overline{\lambda}}{\rightsquigarrow}{}^* \langle P, M'\rangle$, follows, by Lemma 3.3.11, that $\langle \Gamma_P, M\rangle \xrightarrow{\overline{\lambda}}{}^* \langle s, M'\rangle$. From $\langle \Gamma_P, M\rangle$-derivedness of $\langle s, M'\rangle$ and $\langle P, M'\rangle\surd$ follows from Lemma 3.3.12 that $\langle s, M'\rangle \xrightarrow{\varepsilon} \langle \mathsf{skip}, M'\rangle$. By transitivity of $\longrightarrow^*$ follows $\langle \Gamma_P, M\rangle \xrightarrow{\overline{\lambda}\cdot\varepsilon}{}^* \langle \mathsf{skip}, M'\rangle$ and because $\overline{\lambda}$ contains no ε's we have $\widehat{\overline{\lambda}\cdot\varepsilon} = \overline{\lambda}$. □

Using the preceding results, we can show that any output computed by a Gamma program can also be obtained by the corresponding most general schedule. In a sense, this can be seen as showing the completeness of the most general schedule with respect to the corresponding Gamma program.

Theorem 3.3.14 $\forall P, M : \mathcal{C}(P, M) \subseteq \mathcal{C}(\Gamma_P, M)$.

Proof First, note that $\mathcal{C}(P, M) \neq \emptyset$ and $\mathcal{C}(\Gamma_P, M) \neq \emptyset$ for any P and M. We proceed by induction on the structure of P:

- $P = r_1 + \ldots + r_n$: Let $x \in \mathcal{C}(P, M)$ and consider the following cases:
  - $x = \perp$: Hence $\langle P, M\rangle\uparrow$, i.e. if $\langle P, M\rangle = \langle P_0, M_0\rangle$ then for all $i \geq 0$ there exists a σ_i such that $\langle P_i, M_i\rangle \xrightarrow{\sigma_i} \langle P_{i+1}, M_{i+1}\rangle$. By Lemma 3.3.11 follows that, if $\langle \Gamma_P, M\rangle = \langle s_0, M_0\rangle$, then for all $i \geq 0$ there exists a σ_i such that $\langle s_i, M_i\rangle \xrightarrow{\sigma_i} \langle s_{i+1}, M_{i+1}\rangle$. Hence $\perp \in \mathcal{C}(\Gamma_P, M)$.
  - $x = M'$: Hence $\langle P, M\rangle \overset{\overline{\lambda}}{\rightsquigarrow}{}^* \langle P', M'\rangle$ where $\langle P', M'\rangle\surd$ for some M'.
    By Lemma 3.3.13 follows $\langle \Gamma_P, M\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle \mathsf{skip}, M'\rangle$ with $\widehat{\lambda'} = \overline{\lambda}$.
    Hence $M' \in \mathcal{C}(\Gamma_P, M)$.
- $P = P_1 \circ P_2$: Let $x \in \mathcal{C}(P, M)$ and consider the following cases:
  - $x = \perp$: Consider the following cases
    - $\langle P_2, M\rangle\uparrow$: By the induction hypothesis follows $\perp \in \mathcal{C}(\Gamma_{P_2}, M)$. Hence if $\langle \Gamma_{P_2}, M\rangle = \langle s_0, M_0\rangle$, then for all $i \geq 0$ there exists a σ_i such that $\langle s_i, M_i\rangle \xrightarrow{\sigma_i} \langle s_{i+1}, M_{i+1}\rangle$. Then, by (N5) follows that if $\langle \Gamma_{P_2}; \Gamma_{P_1}, M\rangle = \langle s_0, M_0\rangle$, then for all $i \geq 0$ there exists a σ_i such that $\langle s_i, M_i\rangle \xrightarrow{\sigma_i} \langle s_{i+1}, M_{i+1}\rangle$. Hence $\perp \in \mathcal{C}(\Gamma_{P_1 \circ P_2}, M)$.
    - $\langle P_1, M'\rangle\uparrow$ after termination of $\langle P_2, M\rangle$; i.e. $\langle P_2, M\rangle \overset{\overline{\lambda}}{\rightsquigarrow}{}^* \langle P_2, M'\rangle$ where $\langle P_2, M'\rangle\surd$. By Lemma 3.3.13 follows from the latter that $\langle \Gamma_{P_2}, M\rangle \xrightarrow{\overline{\lambda_2'}}{}^* \langle \mathsf{skip}, M''\rangle$. From $\langle P_1, M'\rangle\uparrow$ follows by Lemma 3.3.11 that, if $\langle \Gamma_{P_1}, M\rangle = \langle s_0, M_0\rangle$, then for all $i \geq 0$ there exists a

σ_i such that $\langle s_i, M_i\rangle \xrightarrow{\sigma_i} \langle s_{i+1}, M_{i+1}\rangle$. Then, by (N5) follows that if $\langle \Gamma_{P_2}; \Gamma_{P_1}, M\rangle = \langle s_0, M_0\rangle$, then for all $i \geq 0$ there exists a σ_i such that $\langle s_i, M_i\rangle \xrightarrow{\sigma_i} \langle s_{i+1}, M_{i+1}\rangle$. Hence $\bot \in \mathcal{C}(\Gamma_{P_1 \circ P_2}, M)$.

– $x = M'$: From $M' \in \mathcal{C}(P_1 \circ P_1, M)$ follows $\langle P_1 \circ P_2, M\rangle \overset{\overline{\lambda}}{\rightsquigarrow}{}^* \langle P', M'\rangle$ such that $\langle P', M'\rangle\surd$. The transition sequence can be split into $\langle P_2, M\rangle \overset{\overline{\lambda_2}}{\rightsquigarrow}{}^* \langle P_2', M''\rangle$ where $\langle P_2', M''\rangle\surd$ and $\langle P_1, M''\rangle \overset{\overline{\lambda_1}}{\rightsquigarrow}{}^* \langle P_1', M'\rangle$ where $\langle P_1', M'\rangle\surd$ with $\overline{\lambda} = \overline{\lambda_1} \cdot \overline{\lambda_2}$. Hence $M'' \in \mathcal{C}(P_2, M)$ and $M' \in \mathcal{C}(P_1, M'')$. Then, by the induction hypothesis follows $\langle \Gamma_{P_2}, M\rangle \xrightarrow{\overline{\lambda_2'}}{}^* \langle \mathsf{skip}, M''\rangle$ and $\langle \Gamma_{P_1}, M''\rangle \xrightarrow{\overline{\lambda_1'}}{}^* \langle \mathsf{skip}, M'\rangle$ where $\widehat{\lambda_1'} = \overline{\lambda_1}$ and $\widehat{\lambda_2'} = \overline{\lambda_2}$. Then $\langle \Gamma_{P_2}; \Gamma_{P_1}, M\rangle \xrightarrow{\overline{\lambda_1' \cdot \lambda_2'}}{}^* \langle \mathsf{skip}, M'\rangle$. Hence $M' \in \mathcal{C}(\Gamma_{P_1 \circ P_2}, M)$.

□

3.3.2 Sorts

In the previous section we showed that a most general schedule describes all possible ways in which the corresponding Gamma program may execute and that the most general schedule may terminate in the same multisets as the corresponding program.

In the next section, we show a reverse property: the most general schedule does not describe any execution order that cannot be followed by the corresponding Gamma. This property essentially depends on the fact that any rewrite rule that appears in a $\langle \Gamma_P, M\rangle$-derived configuration is also a rewrite rule of the associated Gamma program.

To reason formally about the rules that appear in a schedule or program we define in this section the notion of *sort*. Furthermore, we show how sorts can be used to simplify showing that the most general schedule can mimic some transitions.

Definition 3.3.15 *The* sort *of a program/schedule is the set of rules that appear in that program/schedule.*

- *The sort function $\mathcal{L}$ for programs is defined inductively by*

$$\begin{aligned} \mathcal{L}(r_1 + \ldots + r_n) &= \{r_1, \ldots, r_n\} \\ \mathcal{L}(P_1 \circ P_2) &= \mathcal{L}(P_1) \cup \mathcal{L}(P_2) \end{aligned}$$

- *For all practical purposes it suffices to reason about the sorts of schedules that use*

a finite number of schedule-identifiers. For this class of schedules, the following construction enables us to determine their sort.

In this definition the set I is used to keep track of schedule-identifiers that have already been encountered. This ensures that the construction is well-defined for recursive schedules (which use a finite number of schedule-identifiers).

$$\mathcal{L}(s) = \mathcal{L}_{\emptyset}(s)$$

where

$$\begin{array}{lcll}
\mathcal{L}_I(\mathsf{skip}) & = & \emptyset & \\
\mathcal{L}_I(r \to s[s']) & = & \{r\} \cup \mathcal{L}_I(s) \cup \mathcal{L}_I(s') & \\
\mathcal{L}_I(s; s') & = & \mathcal{L}_I(s) \cup \mathcal{L}_I(s') & \\
\mathcal{L}_I(s \parallel s') & = & \mathcal{L}_I(s) \cup \mathcal{L}_I(s') & \\
\mathcal{L}_I(c \triangleright s[s']) & = & \mathcal{L}_I(s) \cup \mathcal{L}_I(s') & \\
\mathcal{L}_I(!s) & = & \mathcal{L}_I(s) & \\
\mathcal{L}_I(S(\overline{x})) & = & \mathcal{L}_{I \cup \{S\}}(s) \text{ if } S(\overline{x}) \mathrel{\widehat{=}} s & \text{and } S \notin I \\
\mathcal{L}_I(S(\overline{x})) & = & \emptyset \text{ if } S(\overline{x}) \mathrel{\widehat{=}} s & \text{and } S \in I
\end{array}$$

Example 3.3.16

1. *The sort of the program swap:* $\mathcal{L}(swap) = \{swap\}$.
2. *The sort of a most general schedule* Γ_P *where* $\Gamma_P \mathrel{\widehat{=}} !(r_1 \to \Gamma_P \parallel \ldots \parallel r_4 \to \Gamma_P)$*:* $\mathcal{L}(\Gamma_P) = \{r_1, \ldots r_4\}$.

The operational semantics of schedules (in Figure 3.3) describes how a schedule-term is reduced if the configuration that it is part of makes a transition. Such a reduction of the schedule may decrease, but cannot increase the sort of that schedule.

Lemma 3.3.17 *If* $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$*, then* $\mathcal{L}(s') \subseteq \mathcal{L}(s)$.

Proof Straightforward by transition induction. □

Next, we show that the sort of a Gamma program equals the sort of its most general schedule.

Theorem 3.3.18 $\forall P : \mathcal{L}(P) = \mathcal{L}(\Gamma_P)$

Proof By induction on the structure of P:

- $P = r_1 + \ldots + r_n$: We calculate as follows

$$\begin{aligned}
& \mathcal{L}(\Gamma_P) \\
= & \\
& \mathcal{L}_{\emptyset}(\Gamma_P) \\
= & \\
& \mathcal{L}_{\{\Gamma_P\}}(!(r_1 \to \Gamma_P \parallel \ldots \parallel r_n \to \Gamma_P)) \\
= & \\
& \mathcal{L}_{\{\Gamma_P\}}(r_1 \to \Gamma_P \parallel \ldots \parallel r_n \to \Gamma_P) \\
= & \\
& \bigcup_{i \in \{1,\ldots,n\}}(\{r_i\} \cup \mathcal{L}_{\{\Gamma_P\}(\Gamma_P)}) \\
= & \\
& \{r_1, \ldots, r_n\} \cup \mathcal{L}_{\{\Gamma_P\}}(\Gamma_P) \\
= & \\
& \{r_1, \ldots, r_n\} \\
= & \\
& \mathcal{L}(r_1 + \ldots + r_n)
\end{aligned}$$

- $P = P_1 \circ P_2$: We calculate as follows

$$\begin{aligned}
& \mathcal{L}(\Gamma_{P_1 \circ P_2}) & \\
= & & \text{Definition 3.3.1} \\
& \mathcal{L}(\Gamma_{P_2}; \Gamma_{P_1}) & \\
= & & \\
& \mathcal{L}(\Gamma_{P_2}) \cup \mathcal{L}(\Gamma_{P_1}) & \\
= & & \text{induction hypothesis} \\
& \mathcal{L}(P_2) \cup \mathcal{L}(P_1) & \\
= & & \\
& \mathcal{L}(P_1 \circ P_2) &
\end{aligned}$$

□

In the case studies of Chapter 7 we will regularly need to show that a most general schedule can perform the same transitions as some schedule. Theorem 3.3.20 shows that this is always the case if the sort of the schedule at hand consists of strengthenings of the sort of the most general schedule. The proof of Theorem 3.3.20 is simplified by appealing

to the following property of most general schedules (whose proof is deferred to Chapter 4).

Proposition 3.3.19 *Let P be a simple Gamma program. If $\langle \Gamma_P \,\|\, \Gamma_P, M\rangle \xrightarrow{\lambda} \langle s, M'\rangle$ for some s, then $\exists s' : \langle \Gamma_P, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$.*

Proof Immediate from $\Gamma_P \equiv !\Pi_P$ and Corollary 4.4.23. □

Because sorts are sets of rewrite rules, we can use the generalization of strengthening introduced in Definition 3.1.1 for comparing them: $L_1 \lhd L_2$ reads: the sort L_1 is stronger than L_2 or L_2 is weaker than L_1.

Theorem 3.3.20 *Let P be a simple Gamma program and let s be a schedule such that $\mathcal{L}(s) \lhd \mathcal{L}(P)$. If $\langle s, M\rangle \xrightarrow{\sigma} \langle s', M'\rangle$, then $\exists s'' : \langle \Gamma_P, M\rangle \xrightarrow{\sigma} \langle s'', M'\rangle$*

Proof By transition induction. A transition can be derived in the following ways:

- (N0), then $\lambda = \varepsilon \neq \sigma$. Hence the case holds vacuously.
- (N1), where $s = r \to t_1[t_2]$, from $\langle r \to t_1[t_2], M\rangle \xrightarrow{\sigma} \langle t_1, M\rangle$. Because $r \in \mathcal{L}(s)$ there is some r_i in P such that $r \lhd r_i$. Hence by (N1) $\langle r_i \to \Gamma_P, M\rangle \xrightarrow{\sigma} \langle \Gamma_P, M'\rangle$. Then by (N2), (N6) and (N8), $\langle \Gamma_P, M\rangle \xrightarrow{\sigma} \langle \Pi_{P,i} \,\|\, \Gamma_P, M'\rangle$.
- (N2), where $s = t_1 \,\|\, t_2$, from $\langle t_1, M\rangle \xrightarrow{\lambda} \langle t_1', M'\rangle$. Then $\mathcal{L}(t_1) \subseteq \mathcal{L}(s)$.
  If $\lambda = \sigma$, then the proposition follows immediately from the induction hypothesis. Otherwise, if $\lambda = \varepsilon$, the case holds vacuously.
- (N2), where $s = t_1 \,\|\, t_2$, from $\langle t_2, M\rangle \xrightarrow{\lambda} \langle t_2', M'\rangle$.
  The proof is analogous to the previous case.
- (N3), where $s = t_1 \,\|\, t_2$, from $\langle t_1, M\rangle \xrightarrow{\lambda} \langle t_1', M'\rangle$ and $\langle t_2, M\rangle \xrightarrow{\varepsilon} \langle t_2', M\rangle$. Then $\mathcal{L}(t_1) \subseteq \mathcal{L}(s)$. If $\lambda = \sigma$, then the proposition follows immediately from the induction hypothesis for the former transition. Otherwise, if $\lambda = \varepsilon$, the case holds vacuously.
- (N3), where $s = t_1 \,\|\, t_2$, from $\langle t_1, M\rangle \xrightarrow{\lambda} \langle t_1', M'\rangle$ and $\langle t_2, M\rangle \xrightarrow{\varepsilon} \langle t_2', M\rangle$.
  The proof is analogous to the previous case.
- (N4), where $s = t_1 \,\|\, t_2$, from $\langle t_1, M\rangle \xrightarrow{\sigma_1} \langle t_1', M_1\rangle$ and $\langle t_2, M\rangle \xrightarrow{\sigma_2} \langle t_2', M_2\rangle$ where $M \models \sigma_1 \bowtie \sigma_2$. Then $\mathcal{L}(t_1) \subseteq \mathcal{L}(s)$ and $\mathcal{L}(t_2) \subseteq \mathcal{L}(s)$. Hence by the induction

hypothesis $\langle \Gamma_P, M\rangle \xrightarrow{\sigma_1} \langle s_1, M_1\rangle$ and $\langle \Gamma_P, M\rangle \xrightarrow{\sigma_2} \langle s_2, M_2\rangle$. Because $M \models \sigma_1 \bowtie \sigma_2$, we get by (N4) $\langle \Gamma_P \parallel \Gamma_P, M\rangle \xrightarrow{\sigma} \langle s_1 \parallel s_2, M\rangle$. By Proposition 3.3.19 follows $\langle \Gamma_P, M\rangle \xrightarrow{\sigma} \langle s_3, M'\rangle$ for some s_3.

- (N5), where $s = t_1; t_2$, from $\langle t_1, M\rangle \xrightarrow{\lambda} \langle t_1', M'\rangle$.
  The proof is analogous to that for (N2).

- (N6), where $s = !t$, from $\langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle$.
  Then $\mathcal{L}(t) \subseteq \mathcal{L}(s)$ and the proof proceeds analogous to the case for (N2).

- (N7), where $s = t \parallel !t$, from $\langle t \parallel !t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle$. Then $\mathcal{L}(t) \subseteq \mathcal{L}(s)$, hence $\mathcal{L}(t \parallel !t) \subseteq \mathcal{L}(s)$. The proof proceeds analogous to the case for (N2).

- (N8), where $s = S(\overline{v}) \mathrel{\widehat{=}} t$, from $\langle t[\overline{x} := \overline{v}], M\rangle \xrightarrow{\lambda} \langle t', M'\rangle$. Then $\mathcal{L}(t[\overline{x} := \overline{v}]) = \mathcal{L}(S(\overline{v}))$ and the proof proceeds analogous to the case for (N2).

□

If a most general schedule of a simple program makes a σ-transition, then the schedule arrived at contains at least one instance of the the original most general schedule. Hence, in this way, all schedules that the most general schedule may evolve into are capable of behaving as the original most general schedule.

Lemma 3.3.21 *Let P be a simple program. If $\langle \Gamma_P, M\rangle \xrightarrow{\sigma} \langle s, M'\rangle$, then $s \equiv \Gamma_P \parallel s'$ for some s'.*

Proof Straightforward by transition induction (analogous to the proof of Theorem 3.3.20). □

If the most general schedule contains exactly one rewrite rule, then we can describe more accurately than Lemma 3.3.20 in what form it arrives after a matching a σ-transition.

Lemma 3.3.22 *Let $P = r$ be a simple program and let $s \in \mathbb{S}_{\mathcal{L}(P)}$ be a schedule.*
If $\langle s, M\rangle \xrightarrow{\sigma} \langle s', M'\rangle$, then $\exists s'' : \langle \Gamma_P, M\rangle \xrightarrow{\sigma} \langle s'', M'\rangle$ such that $s'' \equiv \Gamma_P^k$ for $k \geq 1$.

Proof By transition induction – analogous to the proof of Lemma 3.3.20. □

We can say even more about the form of the most general schedule if it has to mimic a single-step transition. In that case, the most general schedule may return to its original form.

Lemma 3.3.23 *Let $P = r$ be a simple program and let $s \in \mathbb{S}_{\mathcal{L}(P)}$ be a schedule. If $\langle s, M\rangle \xrightarrow{\sigma} \langle s', M'\rangle$ is a single-step transition, then $\langle \Gamma_P, M\rangle \xrightarrow{\sigma} \langle \Gamma_P, M'\rangle$.*

Proof By Lemma 3.3.24 follows $\langle r, M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle r, M'\rangle$. By (N1) follows $\langle r \to \Gamma_r, M\rangle \xrightarrow{\sigma} \langle \Gamma_r, M'\rangle$. By (N6) follows $\langle !r \to \Gamma_r, M\rangle \xrightarrow{\sigma} \langle \Gamma_r, M'\rangle$. Then by (N8), (E9) and the definition of Γ_r follows $\langle \Gamma_P, M\rangle \xrightarrow{\sigma} \langle \Gamma_P, M'\rangle$. □

3.3.3 Soundness of the Most General Schedule

In this section we show that a most general schedule does not describe any behaviour that cannot be displayed by the corresponding Gamma program, we first consider this claim for single-step transitions, then multi-step transitions and finally sequences of (multi-step) transitions.

To start, we show that every single-step transition of a schedule is due to the (successful or failing) execution of one particular rewrite-rule from the sort of that schedule.

Lemma 3.3.24 *Let s be a schedule. If $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$ is a single-step transition, then there exists an $r \in \mathcal{L}(s)$ such that*

- *if $\lambda = \varepsilon$, then $\langle r, M\rangle\!\!\downarrow$,*
- *if $\lambda = \sigma$, then $\langle r, M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle r, M'\rangle$.*

Proof By transition induction. □

Now that we have the machinery of sorts at our disposal, we will use it to show that schedules can not make σ-transitions that cannot be mimicked by programs that have the same (or a weaker[1]) sort as that schedule.

Theorem 3.3.25 *Let P be a simple program and s a schedule such that $\mathcal{L}(s) \lhd \mathcal{L}(P)$. If $\langle s, M\rangle \xrightarrow{\sigma} \langle s', M'\rangle$, then $\langle P, M\rangle \overset{\sigma}{\rightsquigarrow} \langle P, M'\rangle$.*

[1]Because sorts are sets of rewrite rules, we can use the generalization of strengthening introduced in Definition 3.1.1 for comparing them: $L_1 \lhd L_2$ reads: the sort L_1 is stronger than L_2 or L_2 is weaker than L_1.

Proof We proceed by induction on the length of the derivation of the transition of s. Consider the possible ways in which the last inference may have been made:

- By (N0): contradicts the σ-label of s's transition.
- By (N1) from $\langle r \to s'[s''], M\rangle \xrightarrow{\sigma} \langle s', M'\rangle$ where $s \equiv r \to s'[s'']$. From $\mathcal{L}(s) \lhd \mathcal{L}(P)$ follows that $P = r' + P'$ for some P' and r' such that $r \lhd r'$. Hence by (C1) follows $\langle r', M\rangle \overset{\sigma}{\rightsquigarrow}_1 \langle r', M'\rangle$. Then by (C3) (and (C2)) follows $\langle P, M\rangle \overset{\sigma}{\rightsquigarrow} \langle P, M'\rangle$.
- By (N2) from $\langle s_1, M\rangle \xrightarrow{\sigma} \langle s_1', M'\rangle$ where $s \equiv s_1 \parallel s_2$. Because $\mathcal{L}(s_1) \subseteq \mathcal{L}(s)$, the result follows immediately by the induction hypothesis.
- By (N3): analogous to the case for (N2).
- By (N4) from $\langle s_1, M\rangle \xrightarrow{\sigma_1} \langle s_1', M_1\rangle$ and $\langle s_2, M\rangle \xrightarrow{\sigma_2} \langle s_2', M_2\rangle$ where $s \equiv s_1 \parallel s_2$ and $M \models \sigma_1 \bowtie \sigma_2$. From $\mathcal{L}(s_1) \subseteq \mathcal{L}(s)$ and $\mathcal{L}(s_2) \subseteq \mathcal{L}(s)$ we get by the induction hypothesis that $\langle P, M\rangle \overset{\sigma_1}{\rightsquigarrow} \langle P, M_1\rangle$ and $\langle P, M\rangle \overset{\sigma_2}{\rightsquigarrow} \langle P, M_2\rangle$. Because $M \models \sigma_1 \bowtie \sigma_2$, we get by (C4) that $\langle P, M\rangle \overset{\sigma}{\rightsquigarrow} \langle P, M'\rangle$.
- By (N5): analogous to the case for (N2).
- By (N6): analogous to the case for (N2).
- By (N7) from $\langle s' \parallel !s', M\rangle \xrightarrow{\sigma} \langle s'', M'\rangle$ where $s \equiv s' \parallel !s'$. Because $\mathcal{L}(s' \parallel !s') \subseteq \mathcal{L}(s)$ the result follows immediately from the induction hypothesis.
- By (N8) from $\langle t, M\rangle \xrightarrow{\sigma} \langle t', M'\rangle$ where $s \equiv t$. The result follows immediately from the induction hypothesis from the fact that $\mathcal{L}(s) \equiv \mathcal{L}(t)$.

□

In general, a Gamma program can mimic all non-ε transitions of a sequence of transitions by a schedule that has a stronger sort.

Corollary 3.3.26 *Let P be a simple program and s a schedule such that $\mathcal{L}(s) \lhd \mathcal{L}(P)$. If $\langle s, M\rangle \xrightarrow{\overline{\lambda}}{}^* \langle s', M'\rangle$, then $\langle P, M\rangle \overset{\overline{\lambda'}}{\rightsquigarrow}{}^* \langle P, M'\rangle$ where $\widehat{\lambda} = \overline{\lambda'}$.*

Proof By induction on the length of the transition sequence.

- $\overline{\lambda} = \langle\ \rangle$: hence $M = M'$ and $s' =' $. By reflexivity of $\rightsquigarrow^*$: $\langle P, M\rangle \overset{\langle\rangle}{\rightsquigarrow}{}^* \langle P, M\rangle$.

- $\overline{\lambda} = \overline{\lambda_1} \cdot \lambda_2$, hence $\langle s, M\rangle \xrightarrow{\overline{\lambda_1}}{}^* \langle s'', M''\rangle$ and $\langle s'', M''\rangle \xrightarrow{\lambda_2} \langle s', M'\rangle$. For the former we get by induction that $\langle P, M\rangle \overset{\overline{\lambda_1'}}{\rightsquigarrow}{}^* \langle P, M''\rangle$ where $\overline{\lambda_1'} = \widehat{\lambda_1}$. For the latter transition, we consider the following cases for λ_2:

  - $\lambda_2 = \varepsilon$: Then $M' = M''$ and $\overline{\lambda_1'} = \widehat{\lambda_1 \cdot \lambda_2}$.
  - $\lambda_2 = \sigma$: By Lemma 3.3.17 follows that $\mathcal{L}(s'') \lhd \mathcal{L}(P)$, hence by Corollary 3.3.28, we get $\langle P, M''\rangle \overset{\lambda_2}{\rightsquigarrow} \langle P, M'\rangle$. By transitivity of $\rightsquigarrow^*$ follows $\langle P, M\rangle \overset{\overline{\lambda_1'} \cdot \lambda_2}{\rightsquigarrow}{}^* \langle P, M'\rangle$ where $\widehat{\lambda} = \overline{\lambda_1'} \cdot \lambda_2$.

□

As corollary of Theorem 3.3.25 we can show that schedules that are built from strengthenings of the rules of a simple program, satisfy the same stable properties as this program.

Lemma 3.3.27 *Let P be a simple program and let s be a schedule. If $\mathcal{L}(s) \lhd \mathcal{L}(P)$ and $[\![q]\!]M$ and stable q, then if $\langle s, M\rangle \xrightarrow{\overline{\lambda}}{}^* \langle s', M'\rangle$, then $[\![q]\!]M'$.*

Proof From $\mathcal{L}(s) \lhd \mathcal{L}(P)$ and $\langle s, M\rangle \xrightarrow{\overline{\lambda}}{}^* \langle s', M'\rangle$ follows by Lemma 3.3.26 that $\langle P, M\rangle \overset{\overline{\lambda'}}{\rightsquigarrow}{}^* \langle P, M'\rangle$ where $\widehat{\lambda} = \overline{\lambda'}$. Then from $[\![q]\!]M$ and the definition of *stable* follows $[\![q]\!]M'$.

□

Lemma 3.3.27 also holds for invariant properties because these are a special case of stable properties.

Another consequence of Theorem 3.3.25 is that a Gamma program P can mimic any σ-transition that is made by a arbitrary $\langle \Gamma_P, M\rangle$-derived configuration.

Corollary 3.3.28 *Let $P = r_1 + \ldots + r_n$ be a simple Gamma program and let $\langle s, M\rangle$ be a $\langle \Gamma_P, M_0\rangle$-derived configuration. If $\langle s, M\rangle \xrightarrow{\sigma} \langle s', M'\rangle$, then $\langle P, M\rangle \overset{\sigma}{\rightsquigarrow} \langle P, M'\rangle$.*

Proof By Lemma 3.3.7 follows that $s \equiv ((r_1 \to \Gamma_P)^{a_1} \parallel \ldots \parallel (r_n \to \Gamma_P)^{a_n}) \parallel {\Gamma_P}^k$ with $a_i \geq 0$ for all i, and $k \geq 0$. Hence $\mathcal{L}(s) \subseteq \mathcal{L}(P)$. Then by Theorem 3.3.25 follows $\langle P, M\rangle \overset{\sigma}{\rightsquigarrow} \langle P, M'\rangle$. □

Lemma 3.3.29 generalizes Corollary 3.3.28 by showing that a simple Gamma program P can mimic any sequence of transitions that a $\langle \Gamma_P, M\rangle$-derived configuration may perform (modulo ε-transitions).

Lemma 3.3.29 *Let P be a simple Gamma program and let $\langle s, M\rangle$ be a $\langle \Gamma_P, M\rangle$-derived configuration. If $\langle s, M\rangle \xrightarrow{\overline{\lambda}}{}^* \langle s', M'\rangle$, then $\langle P, M\rangle \overset{\overline{\lambda'}}{\rightsquigarrow}{}^* \langle P, M'\rangle$ where $\widehat{\lambda} = \overline{\lambda'}$.*

Proof By Lemma 3.3.7 follows that $s \equiv ((r_1 \to \Gamma_P)^{a_1} \parallel \ldots \parallel (r_n \to \Gamma_P)^{a_n}) \parallel {\Gamma_P}^k$ with $a_i \geq 0$ for all i, and $k \geq 0$. Hence $\mathcal{L}(s) \subseteq \mathcal{L}(P)$. Then the result follows by Corollary 3.3.26. □

The preceding results showed that the most general schedule does not describe any behaviour that could not also be the behaviour of the corresponding Gamma program (modulo ε-transitions). Analogously, we show that the most general schedule does not yield any output that could not also be the output of the corresponding Gamma program.

Lemma 3.3.30 *Let $P = r_1 + \ldots + r_n$ be a simple program and let $\langle s, M\rangle$ be a $\langle \Gamma_P, M_0\rangle$-derived configuration. If $\langle s, M\rangle \xrightarrow{\overline{\lambda}}{}^* \langle \mathsf{skip}, M'\rangle$, then $\langle P, M'\rangle\surd$.*

Proof Because $\langle \mathsf{skip}, M'\rangle$ is $\langle \Gamma_P, M_0\rangle$-derived, follows from Lemma 3.3.7 $\langle r_i, M'\rangle\surd$ for all $i : 1 \leq i \leq n$. By (C5) follows $\langle P, M'\rangle\surd$. □

Theorem 3.3.32 shows that the most general schedule only terminates in states that are also final states of the corresponding Gamma program. An important difference with converse Theorem 3.3.14 is that divergence is not covered by this theorem. This discrepancy will be explained below.

As a result of Lemma 3.3.30, it is possible to use the postcondition of a program as the postcondition of the associated most general schedule.

Lemma 3.3.31 *Let $P = r_1 + \ldots + r_n$ be a simple program. If $\langle \Gamma_P, M\rangle \xrightarrow{\overline{\lambda}}{}^* \langle \mathsf{skip}, M'\rangle$, then $\forall i : 1 \leq i \leq n : [\![\dagger r_i]\!]M'$*

Proof From Lemma 2.2.1 and Lemma 3.3.30. □

Theorem 3.3.32 *Let P be a Gamma program and let Γ_P be its most general schedule. For all M: $(\mathcal{C}(\Gamma_P, M) \setminus \{\bot\}) \subseteq \mathcal{C}(P, M)$.*

Proof Note that $\mathcal{C}(\Gamma_P, M) \neq \emptyset$. If $\mathcal{C}(\Gamma_P, M) = \{\bot\}$, then the lemma holds trivially. The remaining cases are dealt with by induction on the structure of program P.

- $P = r_1 + \ldots + r_n$: Suppose $M' \in \mathcal{C}(\Gamma_P, M)$, then $\langle \Gamma_P, M\rangle \xrightarrow{\overline{\lambda}}{}^* \langle \mathsf{skip}, M'\rangle$.
  By Lemma 3.3.29 follows that $\langle P, M\rangle \overset{\overline{\lambda'}}{\rightsquigarrow}{}^* \langle P, M'\rangle$ where $\widehat{\lambda} = \overline{\lambda'}$.
  By Lemma 3.3.30 follows $\langle P, M'\rangle\surd$. Hence $M' \in \mathcal{C}(P, M)$.

- $P = P_1 \circ P_2$: Suppose $M' \in \mathcal{C}(\Gamma_{P_1 \circ P_2}, M)$, then $\langle \Gamma_{P_2}; \Gamma_{P_1}, M\rangle \xrightarrow{\overline{\lambda}}{}^* \langle \mathsf{skip}, M'\rangle$. This transition sequence can be split into $\langle \Gamma_{P_2}, M\rangle \xrightarrow{\overline{\lambda_1}}{}^* \langle \mathsf{skip}, M''\rangle$ and $\langle \Gamma_{P_1}, M''\rangle \xrightarrow{\overline{\lambda_2}}{}^* \langle \mathsf{skip}, M'\rangle$ such that $\overline{\lambda} = \overline{\lambda_1} \cdot \overline{\lambda_2}$. Hence $M'' \in \mathcal{C}(\Gamma_{P_2}, M)$ and $M' \in \mathcal{C}(\Gamma_{P_1}, M'')$. By the induction hypothesis follows $M'' \in \mathcal{C}(P_2, M)$ and $M' \in \mathcal{C}(P_1, M'')$. Hence $\langle P_2, M\rangle \overset{\overline{\mu_1}}{\rightsquigarrow}{}^* \langle P_2', M''\rangle$ with $\langle P_2', M''\rangle\surd$ and $\langle P_1, M''\rangle \overset{\overline{\mu_2}}{\rightsquigarrow}{}^* \langle P_1', M'\rangle$ with $\langle P_1', M'\rangle\surd$. By (C6) and (C7) follows $\langle P_1 \circ P_2, M\rangle \overset{\overline{\mu}}{\rightsquigarrow}{}^* \langle P_1', M'\rangle$ where $\overline{\mu} = \overline{\mu_1} \cdot \overline{\mu_2}$. Hence $M' \in \mathcal{C}(P_1 \circ P_2, M)$.

□

Theorem 3.3.32 does not cover divergence because the most general schedule is always capable of diverging due to its construction using the replication operator. This construction enables the most general schedule to "spawn" an arbitrary number of Π_P-terms while retaining its potential for replication. In this sense, the replication operator corresponds to the exponential operator "!" in linear logic [59] where it can be understood as denoting an infinite resource.

This replication is needed to allow the most general schedule to evolve into an arbitrary – hence possibly dynamically determined – number of rules executing in parallel. However, once these rules are executed in sequence, this introduces the potential of divergence.

An implementor need not worry about this, since there are sensible upper bounds beyond which it is of no use to replicate a (most general) schedule.

1. The number of times a schedule is replicated can be limited by the amount of data that potentially matches the rewrite rules that occur in the replicated schedule. Spawning a higher number of copies can only result in additional ε-transitions which have no effect on the computation.

2. In practice, the number of processors is likely to be a limiting factor. A schedule s in $!s$ may be replicated over all available processors. Replicating a higher number cannot yield a higher degree of parallelism, because there is no means to exploit it.

If a Gamma program terminates at some stage, but its most general schedule continues to spawn, then this will only generate ε-transitions. However, there is also the possibility that the most general schedule diverges, while continuing to make significant (i.e. non ε) transitions. Then, Corollary 3.3.28 (and induction on the number of σ-transitions in the sequence of transitions) assures us that the corresponding program can match this sequence of transitions and also diverge.

Finally, we show that there is no schedule from a given sort whose behaviour is more general than that of the most general schedule for that sort. A consequence of this result, is that there may be other ways of writing schedules which have most general behaviour, but these schedules can not behave in a way that the most general schedule from Definition 3.3.1 can not mimic.

Corollary 3.3.33 *Let P be a simple Gamma program and let s be a schedule such that $\mathcal{L}(s) \lhd \mathcal{L}(P)$. If $\langle s, M\rangle \xrightarrow{\overline{\lambda}}{}^{*} \langle s', M'\rangle$, then $\langle \Gamma_P, M\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle s'', M'\rangle$.*

Proof From $\langle s, M\rangle \xrightarrow{\overline{\lambda}}{}^{*} \langle s', M'\rangle$ and $\mathcal{L}(s) \lhd \mathcal{L}(P)$ follows by Corollary 3.3.26 that $\langle P, M\rangle \overset{\overline{\lambda'}}{\rightsquigarrow}{}^{*} \langle P, M'\rangle$ where $\widehat{\lambda} = \overline{\lambda'}$. Then, by Lemma 3.3.11 follows that $\langle \Gamma_P, M\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle s'', M'\rangle$. □

3.4 Concluding Remarks

In this chapter we showed that the most general schedule serves the purposes for which it is designed:

- The most general schedule describes all possible strategies for executing a Gamma program, but no more.
- The input-output behaviour of a most general schedule matches that of the corresponding Gamma program.

Henceforth, we can use the most general schedule of a Gamma program as initial specification of that program's behaviour.

Acknowledgment

The research described in this chapter was partially funded by ESPRIT project 9102 "Coordination". It was first described in [25] and published as [30].

4 Refinement of Coordination

4.1 Introduction

The Gamma model encourages the programmer to concentrate on *what* he wants to compute, rather than on *how* something should be computed. The resulting Gamma programs are rather abstract; they induce an input-output relation, but leave the actual way in which the output is to be constructed unspecified. Due to their highly nondeterministic character, Gamma programs may be executed in a number of different ways, ranging from the most efficient to the least efficient way.

Next to functionality, efficiency is considered an important aspect of a program. If we are interested in executing a Gamma program efficiently, we must provide additional information about *how* the Gamma program should be executed. To this end, we will use the coordination language that was introduced in Chapter 3 to steer the behaviour of Gamma programs. Hence, specific coordination strategies may be designed which describe efficient ways of executing a Gamma program. However, in designing coordination strategies for Gamma programs we should take care not to invalidate the programs' correctness.

To close the "semantic gap" between the chaotic character of Gamma programs and the imperative method of command required for efficient execution, we develop a formal method for the design of coordination strategies. This method guarantees the correctness of any coordination strategy that is developed using it. Given a Gamma program, this method proceeds as follows:

1. First, construct the most general schedule (as described in Section 3.3) for the given Gamma program. This provides an initial specification of the highly nondeterministic behaviour of the Gamma program in terms of the coordination language.

2. Next, more detailed specifications of the behaviour can be obtained by eliminating nondeterminism from the specification. Specifications of suitable execution strategies can be obtained by repeated reduction of nondeterminacy. The design

formalism only allows the elimination of nondeterminacy from the coordination strategy if its correctness with respect to the Gamma program is preserved.

As an instrument to eliminate nondeterminism, we develop in this chapter several notions of refinement. The problem of finding efficient execution strategies can be broken down into smaller steps by constructing successive refinements where every subsequent refinement gradually achieves more deterministic control.

The refinement of behaviour is depicted in Figure 4.1. The triangles delineate the state-space of a program. The behaviour of a schedule is represented by a (directed) transition graph. Nodes in this graph denote configurations. Initial configurations are marked I and final configurations are marked F. An arrow from one node to another means that the configuration at the tail of the arrow can make a transition which changes the configuration into one represented by the node at the head of the arrow. An execution of a schedule corresponds to a path from the initial node to one of the final nodes. Alternatively, if a schedule does not terminate, an execution corresponds to a cyclic or infinite path through the graph.

In general, schedules are nondeterministic. In the graphical representation, nondeterminism gives rise to branching points. From a branching point execution may proceed along any one of the outgoing arrows. A less ambiguous description of behaviour can be obtained by removing arrows from the graph. In effect, this limits the number of paths from initial to final nodes. This idea forms the basis behind our notion of refinement.

Figure 4.1: Refinement by Limiting Execution Space

The behaviour on the left hand side is a refinement of that on the right hand side. Dotted lines indicate arrows that have been removed. The schedule associated with the behaviour on the left hand side is prevented from exhibiting behaviour that uses the dotted lines.

An important boundary condition on the refinement, is that it maintains total correctness. The refinement suggested by Figure 4.1 can be seen to preserve total correctness, because it retains (at least) one path from an initial to a final node.

The behaviour of a schedule depends on the multiset in which it is executed. If a schedule consists of multiple components that are composed in parallel, then these components share the multiset as state. All the components that run in parallel may modify the state concurrently. A modification of the state by one component may influence the behaviour of another component with which it shares the state. From the perspective of one schedule a modification of the state by another schedule, which might cause the former to behave differently, is called an *interference*.

We develop several notions of refinement based on different assumptions about the possible interferences. These illustrate the effect of particular choices of interference on the strength and usability of the notion of refinement. Subsequently, in Chapter 5, we develop a generalized theory of refinement by parameterizing the possible interference. This theory enables us to design notions of refinement for which interesting properties can be decided by looking at properties of the interference parameter.

The use of these notions of refinement in the derivation of coordination strategies is illustrated by a number of case studies in Chapter 7.

4.2 Refinement based on Simulation

The starting point for our investigations into refinement is the notion of bisimulation. Bisimulation is an equivalence over processes. It equates two processes if they can perform the same actions at corresponding stages of execution.

This notion was successfully used for comparing behaviours of communicating (parallel) processes (see CCS [90]) and automata (see [94]). In order to apply the theory of bisimulation to our setting, we need to make a few modifications.

In CCS [90], process and state are identified, suggesting that every process has a local state which can only be accessed by other processes through message-passing communication. An essential feature of the Gamma model is its use of a shared dataspace. The shared dataspace is a repository of data which may be operated upon concurrently by multiple processes. Every change to the dataspace can be noticed by all processes. Clearly, the behaviour of schedules depends on the shared dataspace. Therefore, we are concerned with the behaviour of configurations $\langle s, M \rangle$ which tie the behaviour of schedules to the contents of the shared dataspace.

Additionally, bisimulation induces an equivalence relation, while we are interested in a partial ordering of refinements which considers a schedule s to be a refinement of a schedule t, if s may engage in a subset of the behaviours of t, but not necessarily the other way around. The notion obtained by breaking the symmetry of bisimulation is studied in Section 4.2.1. In subsequent sections we will improve this notion and study several variations.

4.2.1 Prefix Simulation

The obvious, but as it turns out naive, way of obtaining simulation from bisimulation is by breaking the symmetry. This leads to the following characterization of refinement: s can be simulated by t, if every transition of s can be matched by t. For reasons that will be explained shortly, the notion we have arrived at is called *prefix simulation*.

Definition 4.2.1 *A binary relation on configurations $\mathcal{R} \subseteq \mathbb{C} \times \mathbb{C}$ is a* prefix simulation *if $(\langle s, M\rangle, \langle t, N\rangle) \in \mathcal{R}$ implies, for all λ,*

1. $N = M$
2. $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle \Rightarrow \exists t' : \langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle$ *such that* $(\langle s', M'\rangle, \langle t', M'\rangle) \in \mathcal{R}$

Prefix refinement is defined as the largest prefix simulation relation.

Definition 4.2.2 *Given configurations $\langle s, M\rangle$ and $\langle t, N\rangle$, we say that $\langle s, M\rangle$ is a* prefix refinement *of $\langle t, N\rangle$, written $\langle s, M\rangle \leqslant_p \langle t, N\rangle$, if $(\langle s, M\rangle, \langle t, N\rangle) \in \mathcal{R}$ for some prefix simulation $\mathcal{R}$. This may be equivalently expressed as:*

$$\leqslant_p \; = \bigcup\{\mathcal{R} \mid \mathcal{R} \text{ is a prefix simulation }\}$$

The well-definedness of the relation $\leqslant_p$ can be shown using standard fixed-point techniques (e.g. [90]).

The definition of prefix simulation says that if $\langle s, M\rangle$ is to be a prefix refinement of $\langle t, N\rangle$, then for every transition that $\langle s, M\rangle$ makes, $\langle t, N\rangle$ must be able to follow suit. This works as expected for the following example (we abbreviate $r_i \to \mathsf{skip}$ by r_i).

Example 4.2.3 *Consider the following prefix refinement*

$$\langle r_1; r_2; r_3, M\rangle \leqslant_p \langle r_1 \parallel r_2 \parallel r_3, M\rangle$$

If $\langle r_1; r_2; r_3, M\rangle$ executes its first rule r_1 (resulting in $\langle r_2; r_3, M'\rangle$ for some M') then this can be simulated by $\langle r_1 \parallel r_2 \parallel r_3, M\rangle$ which leads to a configuration $\langle r_2 \parallel r_3, M'\rangle$. Next $\langle r_2; r_3, M'\rangle$ may proceed by executing r_2 yielding $\langle r_3, M''\rangle$ for some M''. This can be mimicked by $\langle r_2 \parallel r_3, M'\rangle$, also ending up as $\langle r_3, M''\rangle$.

We intend to use simulation to repeatedly get successively more refined versions of a schedule. Then in order to retain correctness, it is necessary that a refined schedule terminates in multiset(s) that is (are) also a terminal multiset(s) for the schedule that it refines. The next example illustrates that this requirement is not guaranteed by the notion of prefix simulation.

Example 4.2.4 *We check that the following is a prefix-refinement*

$$\langle r_1, M\rangle \leqslant_p \langle r_1 \parallel r_2, M\rangle$$

If the left hand side executes r_1, it arrives in $\langle \mathsf{skip}, M'\rangle$ for some M'. The right hand side can match execution of r_1 and becomes $\langle r_2, M'\rangle$. Because the refining side $\langle \mathsf{skip}, M'\rangle$ can make no further transition, the definition of $\leqslant_p$ holds vacuously for the remaining configurations. However, the right hand side $\langle r_2, M'\rangle$ has not yet reached a final multiset. Hence in this case the refining side does not reach the same final multiset(s).

From this example we learn that, in general, we have, for any configuration $\langle s, M\rangle$,

$$\langle \mathsf{skip}, M\rangle \leqslant_p \langle s, M\rangle$$

This justifies the refinement of the schedule component of an arbitrary configuration by the empty schedule. This replacement does not in general ensure that the functionality of the schedule is preserved, hence this notion does not satisfy our intended meaning of refinement.

4.3 Strong Statebased Refinement

In the previous section we found out that breaking the symmetry of standard bisimulation does not meet our requirement of preserving total correctness because it allows a refining schedule to terminate prematurely. In this section we set out to remedy this by extending the definition of simulation with an additional condition which states that the refining

(left hand) side may only terminate, if the right hand side may terminate. This leads to the following definition.

Definition 4.3.1 *A binary relation on configurations* $\mathcal{R} \subseteq \mathbb{C} \times \mathbb{C}$ *is a* strong statebased simulation *if* $(\langle s, M\rangle, \langle t, N\rangle) \in \mathcal{R}$ *implies, for all* λ*,*

1. $N = M$
2. $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle \Rightarrow \exists t' : \langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle$ *such that* $(\langle s', M'\rangle, \langle t', M'\rangle) \in \mathcal{R}$
3. $s \equiv \mathsf{skip} \;\Rightarrow t \equiv \mathsf{skip}$

In compliance with [90], this notion of simulation is called *strong* simulation because every single transition of the refining schedule is matched by a single transition of the refined schedule. In Section 4.3.3 we shall relax this property by introducing a weak notion of refinement. The adjective *statebased* is added, because the current state, represented by the multiset, is taken into account – this in contrast to the notion presented in Section 4.4.

We show some basic properties of strong statebased simulation.

Proposition 4.3.2 *Let* $\mathcal{R}_i$ *for* $i = 1, 2, \ldots$ *be strong statebased simulations. Then the following are also strong statebased simulations*

1. *the identity relation on configurations:* $Id_{\mathbb{C}} = \{(\langle s, M\rangle, \langle s, M\rangle) \mid \langle s, M\rangle \in \mathbb{C}\}$,
2. *the composition:* $\mathcal{R}_1\mathcal{R}_2$,
3. *the union:* $\bigcup_{i\in I} \mathcal{R}_i$.

Proof Postponed to Section 5.2. □

Let $\langle s, M\rangle$ and $\langle t, M\rangle$ be configurations. We say that $\langle s, M\rangle$ is a *strong statebased refinement* of $\langle t, M\rangle$, denoted $\langle s, M\rangle \leqq \langle t, M\rangle$, if $(\langle s, M\rangle, \langle t, M\rangle) \in \mathcal{R}$ for some strong statebased simulation $\mathcal{R}$. Hence, we define the strong statebased refinement relation as the maximal strong statebased simulation. Strong statebased equivalence, denoted $\cong$, is defined as the intersection of strong statebased refinement and its inverse.

Definition 4.3.3

1. $\leqq \; = \bigcup\{\mathcal{R} \mid \mathcal{R}$ *is a strong statebased simulation* $\}$
2. $\cong \; = \; \leqq \cap \leqq^{-1}$

As a notational convenience[1], we write $s \leqq_M t$ iff $\langle s, M\rangle \leqq \langle t, M\rangle$. This allows us to consider $\leqq_M$ as a binary relation on schedules.

Proposition 4.3.4

1. $\leqq$ *is the largest strong statebased simulation.*
2. $\leqq$ *is a partial order.*
3. $\cong$ *is an equivalence relation.*

Proof Postponed to Section 5.2. □

In Section 5.2 we will show that $\leqq$ is the largest relation that satisfies the definition of strong statebased simulation. Hence $\leqq$ defines the relation that contains precisely all strong statebased simulations.

To establish $\langle s, M\rangle \leqq \langle t, N\rangle$ it suffices to devise a relation $\mathcal{R}$, such that $(\langle s, M\rangle, \langle t, N\rangle) \in \mathcal{R}$, and prove that $\mathcal{R}$ is a strong statebased simulation relation. From the fact that $\leqq$ is the largest statebased simulation follows for any such simulation relation $\mathcal{R}$, that $\mathcal{R} \subseteq \leqq$, hence $\langle s, M\rangle \leqq \langle t, N\rangle$.

Figure 4.2 shows a Hasse diagram that illustrates the notion of refinement implied by strong statebased simulation. An arc from a node v to a node u indicates that the schedule in u (represented by the possible executions) is a refinement of the schedule in v. A dotted arc from v to u indicates that the schedule in v is a prefix-refinement of the schedule in u. For simplicity we have omitted the multiset component from the configurations in this figure.

In [90] Milner introduces a generalization of bisimulation, called bisimulation *up-to*, that is often somewhat easier to use than plain bisimulation because it allows simpler formulation of simulation relations. This *up-to* method and its advantages carry over straightforwardly to strong statebased refinement. We develop this theory next.

We write $\leqq \mathcal{R} \leqq$ to denote the composition of binary relations, so that $\langle s, M\rangle \leqq \mathcal{R} \leqq \langle s', M'\rangle$ means that there are some $\langle t, M\rangle$ and $\langle t', M'\rangle$ such that $\langle s, M\rangle \leqq \langle t, M\rangle$, $\langle t, M\rangle \mathcal{R} \langle t', M'\rangle$ and $\langle t', M'\rangle \leqq \langle s', M'\rangle$.

Definition 4.3.5 *A relation* $\mathcal{R} \subseteq \mathbb{C} \times \mathbb{C}$ *is a* strong statebased simulation up-to $\leqq$ *if* $(\langle s, M\rangle, \langle t, N\rangle) \in \mathcal{R}$ *implies, for all* λ*,*

[1] Formally, this notation introduces a family of refinement relations – one for every multiset.

Figure 4.2: *Hasse diagram of the refinements of* $r_1 \parallel r_2$.

1. $M = N$
2. $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle \Rightarrow \exists t' : \langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle$ *such that* $\langle s', M'\rangle \leqq \mathcal{R} \leqq \langle t', M'\rangle$
3. $s \equiv \mathsf{skip} \Rightarrow t \equiv \mathsf{skip}$

Proposition 4.3.6
If $\mathcal{R}$ *is a strong statebased simulation up-to* $\leqq$, *then* $\mathcal{R} \subseteq \leqq$.

Proof Postponed to Section 5.2. □

The next example illustrates how strong statebased simulation can be used to verify that one schedule "correctly implements" another. Furthermore, it shows how the up-to technique may simplify the simulation relation.

Example 4.3.7 *Let* r_1 *and* r_2 *be rules, then*

$$\langle (r_1; r_2) \parallel (r_2; r_1), M_1\rangle \leqq \langle !(r_1 \parallel r_2), M_1\rangle$$

In order to show this consider the following relation $\mathcal{R}$:

$$
\begin{array}{rcll}
\mathcal{R} & = & \{(\langle (r_1; r_2) \parallel (r_2; r_1), M_1\rangle, \langle !(r_1 \parallel r_2), M_1\rangle)\} & (1)\\
 & \cup & \{(\langle r_2 \parallel (r_2; r_1), M_2\rangle, \langle r_2 \parallel (r_2 \parallel r_1), M_2\rangle)\} & (2)\\
 & \cup & \{(\langle (r_1; r_2) \parallel r_1, M_3\rangle, \langle r_1 \parallel (r_1 \parallel r_2), M_3\rangle)\} & (3)\\
 & \cup & \{(\langle r_1 \parallel r_2, M_4\rangle, \langle r_1 \parallel r_2, M_4\rangle)\} & (4)\\
 & \cup & \{(\langle r_2; r_1, M_4\rangle, \langle r_1 \parallel r_2, M_4\rangle)\} & (5)\\
 & \cup & \{(\langle r_1; r_2, M_4\rangle, \langle r_1 \parallel r_2, M_4\rangle)\} & (6)\\
 & \cup & \{(\langle r_1, M_5\rangle, \langle r_1, M_5\rangle)\} & (7)\\
 & \cup & \{(\langle r_2, M_6\rangle, \langle r_2, M_6\rangle)\} & (8)\\
 & \cup & \{(\langle \mathsf{skip}, M_7\rangle, \langle \mathsf{skip}, M_7\rangle)\} & (9)
\end{array}
$$

By considering the possible transitions for each of the elements of $\mathcal{R}$, it follows that $\mathcal{R}$ is a strong statebased simulation. We depict the (relevant parts of the) transition graphs of these schedules in Figure 4.3. Note that the numbers used to distinguish subsets of $\mathcal{R}$ correspond to the different states of the computation.

Figure 4.3: Transition graphs of $(\mathtt{r_1}; \mathtt{r_2}) \parallel (\mathtt{r_2}; \mathtt{r_1})$ (left) and partially of $!(\mathtt{r_1} \parallel \mathtt{r_2})$ (right).

Later on in this thesis (Section 4.4) we will find out that $\langle r_1; r_2, M\rangle \leqq \langle r_1 \parallel r_2, M\rangle$ and $\langle r_2; r_1, M\rangle \leqq \langle r_1 \parallel r_2, M\rangle$. Hence $\langle r_1; r_2, M\rangle \leqq \langle r_1 \parallel r_2, M\rangle \mathcal{R} \langle r_1 \parallel r_2, M\rangle \leqq \langle r_1 \parallel r_2, M\rangle$ (and similarly for $r_2; r_1$). This can be used to show that the relation $\mathcal{R}' = (1) \cup (2) \cup (3) \cup$

$(4) \cup (7) \cup (8) \cup (9)$ *(i.e. components (5) and (6) can be omitted) is a strong statebased simulation up–to* $\leqq$*. From this example we see that considering simulation up–to* $\leqq$ *reduces the complexity of refinement proofs.*

An important feature of the current notion of refinement is its ability to exploit properties of the multiset. This is possible because the multiset is an explicit component of the simulation relation. The following example illustrates the idea.

Example 4.3.8 *Consider a Gamma program for computing the sum of a multiset of numbers:*

$$add \mathrel{\widehat{=}} x, y \mapsto x + y \Leftarrow true$$

The program operates by adding pairs of numbers from the initial multiset in any order (hence possibly in parallel). The possible behaviours of this program are described by its most general schedule:

$$\Gamma_{add} \mathrel{\widehat{=}} !(add \to \Gamma_{add})$$

A schedule that executes the rule add n times in sequence (hence is more deterministic than the most general schedule) is given by $Sum(n+1)$ *where*

$$Sum(i) \mathrel{\widehat{=}} i > 1 \triangleright (add; Sum(i-1))$$

In order for the schedule $Sum(i)$ *to correctly compute the sum of some multiset* M*, the parameter* i *must reflect the number of elements in* M*. In relation* $\mathcal{R}$ *(4.1), the link between the number of elements of* M *and the parameter* i *is made using the condition* $i = \#M + 1$*.*

$$\mathcal{R} = \{(\langle Sum(i), M\rangle, \langle \Gamma_{add}, M\rangle) \mid i = \#M + 1,\ \#M > 0\} \tag{4.1}$$

It is straigthforward to show that $\mathcal{R}$ *is a strong statebased simulation. From this we conclude* $\langle Sum(n), M\rangle \leqq \langle \Gamma_{add}, M\rangle$ *where* $n = \#M + 1$ *for all* $\#M > 0$*.*

Because computing the sum of n numbers requires $n-1$ additions and the schedule $Sum(i)$ performs $i-1$ additions, one would expect that

$$\langle Sum(n), M\rangle \leqq \langle \Gamma_{add}, M\rangle \text{ with } n = \#M \tag{4.2}$$

However, the following counterexample disproves this refinement. Consider, for instance, the initial multiset $\{1, 5, 3\}$. Then $Sum(3)$ may perform the following transition

sequence.

$$\langle Sum(3), \{1,5,3\}\rangle \overset{\{6\}/\{1,5\}}{\longrightarrow} \langle Sum(2), \{6,3\}\rangle \overset{\{9\}/\{6,3\}}{\longrightarrow} \langle \mathsf{skip}, \{9\}\rangle$$

The most general schedule Γ_{add} can also make these transitions, but inevitably needs an additional ε-transition to detect termination, e.g.

$$\langle \Gamma_{add}, \{1,5,3\}\rangle \overset{\{6\}/\{1,5\}}{\longrightarrow} \langle \Gamma_{add}, \{6,3\}\rangle \overset{\{9\}/\{6,3\}}{\longrightarrow} \langle \Gamma_{add}, \{9\}\rangle \overset{\varepsilon}{\longrightarrow} \langle \mathsf{skip}, \{9\}\rangle$$

This discrepancy is compensated for by letting the *Sum* schedule execute an additional (final) *add* rule (which will, just like the most general schedule, always be ε).

This solution is rather ad-hoc. There are two reasons for which we would like to consider Equation (4.2) from Example 4.3.8 to be a valid refinement.

- The trailing ε transition is an artifact of the semantics of schedules. The ε-label is used to distinguish failing from successful execution of a rewrite rule. The most general schedule has no knowledge about the contents of the multiset and interprets the ε transition as the signal that there is no opportunity to execute a rule. If the program, and hence its most general schedule, consists of a single rewrite rule, then the failure of this rule indicates that a final state has been reached.

- If the behaviour of a configuration $\langle s, M\rangle$ differs from that of another configuration $\langle t, M\rangle$ only by the fact that they make a different number of ε-transitions, we still want to consider them equivalent because ε transitions do not change the multiset, hence do not change the functionality of a schedule.

□

In Section 4.3.3 we propose a more liberal notion of refinement that supports the intuition of the latter reason and thereby allows a more elegant solution.

4.3.1 Soundness of Strong Statebased Refinement

In [65] Hankin et al. define a *capability* function which models the input-output behaviour of Gamma programs. Subsequently, they use the relational ordering (subset inclusion) of the set of possible outcomes as the basis of a calculus of refinement. In this section we show that strong statebased refinement preserves the relational ordering on schedules; i.e. if s is a statebased refinement of t, then the set of outputs of s is a subset of the set of outputs of t.

Theorem 4.3.9 *If* $\langle s, M\rangle \leqq \langle t, M\rangle$, *then* $\mathcal{C}(s, M) \subseteq \mathcal{C}(t, M)$.

Proof First observe that $C(s, M) \neq \emptyset$ for all s, M. Assume $x \in \mathcal{C}(s, M)$. We show that $x \in \mathcal{C}(t, M)$.
Consider the following cases:

- $x = \perp$:
  Hence if $\langle s, M\rangle = \langle s_0, M_0\rangle$, then for all $i \geq 0$ there exists a λ_i such that $\langle s_i, M_i\rangle \xrightarrow{\lambda_i} \langle s_{i+1}, M_{i+1}\rangle$. By $\langle s, M\rangle \leqq \langle t, M\rangle$ follows $\langle t, M\rangle = \langle t_0, M_0\rangle$ such that for all $i \geq 0$ that there exists a λ_i such that $\langle t_i, M_i\rangle \xrightarrow{\lambda_i} \langle t_{i+1}, M_{i+1}\rangle$. Hence $\perp \in \mathcal{C}(t, M)$.

- $x = M'$:
  Hence $\langle s, M\rangle \xrightarrow{\overline{\lambda}}{}^{*} \langle \mathsf{skip}, M'\rangle$. By $\langle s, M\rangle \leqq \langle t, M\rangle$ and induction on the length of the transition sequence, follows $\langle t, M\rangle \xrightarrow{\overline{\lambda}}{}^{*} \langle \mathsf{skip}, M'\rangle$. Hence $M' \in \mathcal{C}(t, M)$.

□

4.3.2 Compositionality Issues of Statebased Refinement

A method of reasoning about programs is called *compositional* if properties of a program as a whole can be inferred from properties about the individual components of a program.

Compositional reasoning allows one to focus on one part of a program without having to take its context into account. Conversely, non-compositional methods of reasoning are tedious to use for large programs because this requires that programs be considered in their entirety.

A common approach, followed for instance by Milner [90], is to define an equivalence relation over programs (in terms of their semantics) and show that this relation is a congruence over program terms. The congruence property makes it possible to use program equivalences as equational laws to reason about programs in a modular (or compositional) fashion. Equational reasoning facilitates formal calculation and avoids the complexity of operational details.

Attempts at obtaining a compositional method of reasoning about statebased refinement according to this approach, run into two kinds of problems. We illustrate these by looking at parallel and sequential composition.

Compositionality of Parallel Composition

Compositional reasoning about statebased refinement of schedules could be justified by showing that statebased refinement is preserved by all combinators from the coordination language. For parallel composition we would need to show the following (where we write $s_1 \leqq_M t_1$ for $\langle s_1, M\rangle \leqq \langle t_1, M\rangle$ because this highlights that $\|$ is a combinator for schedules rather than configurations):

$$\text{if } s_1 \leqq_M t_1 \text{ and } s_2 \leqq_M t_2 \text{ then } s_1 \,\|\, s_2 \leqq_M t_1 \,\|\, t_2$$

However, the next counterexample shows that this statement is false.

Example 4.3.10 *Consider the following schedules*

$$\begin{array}{lcll} Dec & \widehat{=} & (dec \to Dec) & \text{where} \quad dec \quad x \mapsto x-1 \Leftarrow x > 0 \\ Inc & \widehat{=} & (inc \to Inc) & \qquad\quad inc \quad x \mapsto x+1 \Leftarrow x < 2 \end{array}$$

Then, for the initial multiset $M_0 = \{0\}$, the following refinements hold:

$$dec \leqq_{M_0} Dec$$

and

$$inc; inc; inc \leqq_{M_0} Inc$$

Next, we show that $dec \,\|\, inc; inc; inc \not\leqq_{M_0} Dec \,\|\, Inc$.

Execution of $\langle Dec \,\|\, Inc, M_0\rangle$ may start with the execution of a rewrite rule dec. This rewrite fails which yields the configuration $\langle Inc, M_0\rangle$. This configuration terminates once it reaches $\langle \mathsf{skip}, \{2\}\rangle$. Alternatively, execution of $\langle Dec \,\|\, Inc, M_0\rangle$ may reach a multiset $\{2\}$ by repeated execution of inc. Then Inc may reduce to skip *after which Dec continues execution until the multiset $\{0\}$ is reached. Hence if $\langle Inc \,\|\, Dec, M_0\rangle$ terminates, the multiset equals either $\{0\}$ or $\{2\}$.*

In contrast, the execution of the configuration $\langle dec \,\|\, (inc; inc; inc), \{0\}\rangle$ may terminate in one of the multisets $\{1\}$ or $\{2\}$. Because $Dec \,\|\, Inc$ can not terminate in $\{1\}$, it is not *refined by the schedule $dec \,\|\, (inc; inc; inc)$.*

Hence, the statebased refinement relation does not hold for the parallel composition of the schedules because the interaction between *dec* and $inc; inc; inc$ can give rise to behaviours of $dec \,\|\, inc; inc; inc$ (the composition of refined schedules) that can not be displayed by $Dec \,\|\, Inc$ (the composition of the original schedules).

In general, statebased refinement is not preserved by parallel composition because the interaction of the refined components of the composition may give rise to behaviour that is not taken into account by the refinements of the individual schedules.

Compositionality of Sequential Composition

In order to reason compositionally about statebased refinement of sequentially composed schedules, we need to complete the compositionality-formula below such that it yields a valid statement. The question mark in this formula indicates a position where some multiset M needs to be substituted.

$$\text{if } s_1 \leqq_M t_1 \text{ and } s_2 \leqq_? t_2 \text{ then } s_1; s_2 \leqq_M t_1; t_2$$

We consider the possibilities for choosing a multiset to place at the question mark.

From an purely mathematical point of view, the only sensible choice is to relate s_2 and t_2 by the same relation as that which relates s_1 and t_1 and their composition - hence choose "M" (because precongruence is a property of a single relation). However, the statement thus obtained is false: the fact that t_2 can simulate s_2 in M, does not guarantee that t_2 can simulate s_2 after execution of s_1, because execution of s_1 in M will generally change the multiset into something other than M which may cause s_2 to behave in a completely different manner compared to how it would behave when started in M. We have no information about whether t_2 can simulate s_2 starting in a multiset that is different from M.

The aspect that prevents precongruence for sequential composition is analogous to that what prevented precongruence for parallel composition: One of the components of the (in this case sequential) composition modifies the multiset which may cause the composition of the refined schedules to behave in a way that was not taken into account by the refinements of the individual schedules.

For the case of sequential composition it is always the left-hand side (first) component that modifies what would have been the starting multiset of the right-hand side (second) component. In the case of parallel composition, the order of interference is arbitrary.

The preceding argumentation identifies the need to know in which multiset execution of s_1 terminates and hence execution of s_2 starts. This brings up two problems: Firstly, the outcome of a schedule may be nondeterministic, hence all possible outcomes would need to be considered. Secondly, checking that the multiset is an outcome of s_1 requires the use of an additional proof method. This could complicate practical use.

Even though the method suggested may not be practical in general, it will be worthwhile to develop it a little bit further because it may provide a method of last resort when other more practical methods fail (which will turn out to be the case in Chapter 7).

Lemma 4.3.12 suggests a method of reasoning about statebased refinement of sequentially composed schedules. It requires that the schedule s_2 is a refinement of t_2 for all possible outcomes of s_1. Lemma 4.3.12 uses the auxiliary result of Lemma 4.3.11 which shows that the set of possible outcomes may only decrease as execution proceeds.

Lemma 4.3.11 *If* $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$, *then* $\mathcal{C}(s', M') \subseteq \mathcal{C}(s, M)$.

Proof Straightforward from Definition 3.2.3.

□

Lemma 4.3.12 *If*

1. $s_1 \leqq_M t_1$,
2. $\forall M' \in \mathcal{C}(s_1, M) : s_2 \leqq_{M'} t_2$

then $s_1; s_2 \leqq_M t_1; t_2$.

Proof Let $\mathcal{R} = \{(\langle s_1; s_2, M\rangle, \langle t_1; t_2, M\rangle) \mid s_1 \leqq_M t_1 \ \wedge \ \forall M' \in \mathcal{C}(s_1, M) : s_2 \leqq_{M'} t_2\}$. We show that $\mathcal{R}$ is a strong statebased simulation up-to $\leqq$.
transition

- Assume $\langle s_1, M\rangle \xrightarrow{\lambda} \langle s_1', M'\rangle$.
  Then from $s_1 \leqq_M t_1$ follows $\langle t_1, M\rangle \xrightarrow{\lambda} \langle t_1', M'\rangle$ such that $s_1' \leqq_{M'} t_1'$. By $(N5)$ follows $\langle t_1; t_2, M\rangle \xrightarrow{\lambda} \langle t_1'; t_2, M'\rangle$. By Lemma 4.3.11 follows $\mathcal{C}(s_1', M') \subseteq \mathcal{C}(s_1, M)$. Hence $\forall M'' : M'' \in \mathcal{C}(s_1', M') : s_2 \leqq_{M'} t_2$. By reflexivity of $\leqq$ follows $(\langle s_1'; s_2, M'\rangle, \langle t_1'; t_2, M'\rangle) \in \leqq \mathcal{R} \leqq$.

- Assume $s_1 \equiv \mathsf{skip}$ and $\langle s_2, M\rangle \xrightarrow{\lambda} \langle s_2', M'\rangle$.
  From $s_1 \leqq_M t_1$ follows $t_1 \equiv \mathsf{skip}$. Then from $s_2 \leqq_M t_2$ follows $\langle t_2, M\rangle \xrightarrow{\lambda} \langle t_2', M'\rangle$ such that $s_2' \leqq_{M'} t_2'$. From $\mathsf{skip}; s \cong_M s$ and $s \cong_M s; \mathsf{skip}$ follows $\langle s_2', M'\rangle \leqq \langle \mathsf{skip}; s_2', M'\rangle \mathcal{R} \langle \mathsf{skip}; t_2', M'\rangle) \leqq \langle t_2', M'\rangle$.

termination
$s_1; s_2 \equiv \mathsf{skip}$ only if $s_1 \equiv \mathsf{skip}$ and $s_2 \equiv \mathsf{skip}$. From $s_1 \equiv \mathsf{skip}$ and $s_1 \leqq_M t_1$ follows

$t_1 \equiv \mathsf{skip}$. From $s_2 \equiv \mathsf{skip}$ and $s_2 \leqq_M t_2$ follows $t_2 \equiv \mathsf{skip}$. Hence $t_1; t_2 \equiv \mathsf{skip}$. □

The main issue that Lemma 4.3.12 deals with is the input of the right component (which is the output of the left component). Refining only the left argument of a sequential composition is more straightforward.

Corollary 4.3.13 *For all t, if $s' \leqq_M s$ then $s'; t \leqq_M s; t$.*

Proof By Lemma 4.3.12. □

The approach suggested by Lemma 4.3.12 allows modular substitution which is typical of compositional methods of reasoning. However, the approach is not compositional: in order to refine the subterm s_2 of $s_1; s_2$ knowledge about the context (i.e. the outcome of s_1) is used. Hence the practical use of this method is limited by the ease by which the set of outcomes $\mathcal{C}(s_1)$ can be determined and the ease by which the set of outcomes of the sequential composition $\mathcal{C}(s_1; s_2)$ can be determined given the input-output behaviour of the constituents $\mathcal{C}(s_1)$ and $\mathcal{C}(s_2)$. Hence, we have reduced the problem of finding a method for reasoning compositionally about statebased refinement of behaviour to finding a compositional method for reasoning about the capability of schedules.

4.3.3 Weak Statebased Refinement

Example 4.3.8 prompted the observation that strong statebased refinement does not justify refinements where the only difference between configurations is the number of ε-steps they may make. From the semantic rules in Figure 3.3 follows that ε-transitions do not change the multiset. So adding (or removing) ε-transitions to (from) a transition sequence cannot change the outcome (hence functionality) of a computation.

Analogously to [90] this brings us to define a notion of refinement, that is insensitive to ε-transitions. This notion is called *weak* because it allows a single transition from one configuration to be matched by a sequence of (zero or more) transitions by the other configuration – provided they have the same effect on the multiset.

Recall that $\xrightarrow{\overline{\lambda}}{}^*$ denotes the reflexive transitive closure of the transition relation $\longrightarrow$ from Figure 3.3. The label $\overline{\lambda}$ denotes the sequence obtained by concatenating, in order, all individual labels of the constituent transitions. For convenience, we identify the singleton sequence $\langle\, \lambda \,\rangle$ with its only element λ. Furthermore, we use $\widehat{\lambda}$ to denote the sequence $\overline{\lambda}$ where all occurrences of ε have been removed.

Definition 4.3.14 *A relation* $\mathcal{R} \subseteq \mathbb{C} \times \mathbb{C}$ *is a* weak statebased simulation *if, for all* $(\langle s, M\rangle, \langle t, N\rangle) \in \mathcal{R}$*, for all* λ

1. $M = N$
2. $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle \Rightarrow \exists t' : \langle t, M\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle t', M'\rangle$ *such that* $(\langle s', M'\rangle, \langle t', M'\rangle) \in \mathcal{R}$ *and* $\overline{\lambda'} = \varepsilon^k \cdot \widehat{\lambda}$ *for some* $k \geq 0$
3. $s \equiv \mathsf{skip} \;\Rightarrow \langle t, M\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle \mathsf{skip}, M\rangle$ *where* $\widehat{\lambda'} = \langle\ \rangle$

Proposition 4.3.15 *Let* $\mathcal{R}_i$ *for* $i = 1, 2, \ldots$ *be weak statebased simulations. Then the following are also weak statebased simulations*

1. *the identity relation on configurations:* $Id_{\mathbb{C}} = \{(\langle s, M\rangle, \langle s, M\rangle) \mid \langle s, M\rangle \in \mathbb{C}\}$,
2. *the composition:* $\mathcal{R}_1\mathcal{R}_2$,
3. *the union:* $\bigcup_{i\in I} \mathcal{R}_i$.

Proof Postponed to Section 5.5. □

Let $\langle s, M\rangle$ and $\langle t, M\rangle$ be configurations. We say that $\langle s, M\rangle$ is a *weak statebased refinement* of $\langle t, M\rangle$, denoted $\langle s, M\rangle \gtrapprox \langle t, M\rangle$, if $(\langle s, M\rangle, \langle t, M\rangle) \in \mathcal{R}$ for some weak statebased simulation $\mathcal{R}$. As is standard, we define weak statebased equivalence, denoted $\approx$, as the kernel of weak statebased refinement.

Definition 4.3.16

1. $\lessapprox = \bigcup\{\mathcal{R} \mid \mathcal{R}$ *is a weak statebased simulation* $\}$
2. $\approx \;=\; \lessapprox \cap \gtrapprox^{-1}$

Proposition 4.3.17

1. $\lessapprox$ *is the largest weak statebased simulation.*
2. $\lessapprox$ *is a partial order.*
3. $\approx$ *is an equivalence relation.*

Proof Postponed to Section 5.5. □

Analogously to strong statebased refinement, we prove in Section 5.5 that $\precsim$ is the largest relation that satisfies the definition of weak statebased simulation. Hence $\precsim$ defines the relation that contains precisely all weak statebased simulations.

We briefly explain that we defined weak statebased simulation such that it is insensitive to a differing number of ε transitions. To this end, we expound how ε-transitions made by either the schedule that is being refined (t) or by the refining schedule (s) may be disregarded.

- A transition $\langle s, M\rangle \xrightarrow{\varepsilon} \langle s', M\rangle$ by s may be matched by a sequence $\langle t, M\rangle \xrightarrow{\langle\,\rangle}{}^{*} \langle t, M\rangle$ of zero transitions by t. Hence, this allows the refining schedule s to make more ε transitions than the schedule t that is being refined (at any stage of execution of t).

- A sequence of transitions $\langle t, M\rangle \xrightarrow{\varepsilon^k\cdot\lambda}{}^{*} \langle t', M'\rangle$ by t may be matched by a single transition $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$ by s. This allows all ε-transitions, made by t, that precede a λ-transition to be skipped by s.

  The elimination of ε-transitions that are made by t following a λ-transition, is justified by clause 3 of Definition 4.3.14.

In [90], Milner uses a symmetrical way of disregarding "silent"-labels[2]: an action λ may be matched by a sequence of transitions with consecutive labels $\varepsilon^{k_1}\cdot\widehat{\lambda}\cdot\varepsilon^{k_2}$. In Definition 4.3.14 we use the (asymmetrical) condition $\overline{\lambda} = \varepsilon^k \cdot \widehat{\lambda}$ in clause 2 which only allows the elimination of ε-labels before the λ. As illustrated above, these definitions are effectively the same. However, the asymmetrical way of defining the removal of ε's provides a clearer separation between the functions of the second and third clause of Definition 4.3.14 and this turned out to be a better structure for proving precongruence of variants of weak simulation (which are developed in subsequent chapters).

We continue by developing the *up-to* technique (from [90]) for weak statebased refinement.

Definition 4.3.18 *A relation* $\mathcal{R} \subseteq \mathbb{C} \times \mathbb{C}$ *is a* weak statebased simulation up-to $\precsim$ *if, for all* $(\langle s, M\rangle, \langle t, N\rangle) \in \mathcal{R}$

[2]To emphasize the analogy we write ε for Milner's τ and λ for an action.

1. $M = N$
2. $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle \Rightarrow \exists t' : \langle t, M\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle t', M'\rangle$ *such that* $\langle s', M'\rangle \precapprox \mathcal{R} \precapprox \langle t', M'\rangle$ *and* $\overline{\lambda'} = \varepsilon^k \cdot \widehat{\lambda}$ *for some* $k \geq 0$
3. $s \equiv \mathsf{skip} \;\Rightarrow \langle t, M\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle \mathsf{skip}, M\rangle$ *where* $\widehat{\lambda'} = \langle\ \rangle$

Proposition 4.3.19 *If* $\mathcal{R}$ *is a weak statebased simulation up-to* $\precapprox$, *then* $\mathcal{R} \subseteq \precapprox$.

Proof Postponed to Section 5.5. □

Using the weak notion of simulation we are now able to prove the refinement from Example 4.3.8.

Example 4.3.20 *Let* $\mathcal{R} = \{(\langle Sum(n), M\rangle, \langle \Gamma_{add}, M\rangle) \mid \#M = n, n \geq 0\}$.
We prove that $\mathcal{R}$ *is a weak statebased simulation by induction on* n.
Proof

- $n \leq 1$*: then* $Sum(n) \equiv \mathsf{skip}$. *Because* $\#M \leq 1$ *we derive by* $(N0)$, $(N6)$ *and* $(N9)$, $\langle \Gamma_{add}, M\rangle \xrightarrow{\varepsilon} \langle \mathsf{skip}, M\rangle$. *By definition of* $\longrightarrow^*$ *follows* $\langle \Gamma_{add}, M\rangle \xrightarrow{\varepsilon}{}^{*} \langle \mathsf{skip}, M\rangle$.
- $n > 1$ *and* $\langle Sum(n), M\rangle \xrightarrow{\sigma} \langle Sum(n-1), M'\rangle$ *where* $\#M' = n - 1$.
  Then, by $(N1)$, $(N6)$ *and* $(N9)$ *follows* $\langle \Gamma_{add}, M\rangle \xrightarrow{\sigma} \langle \Gamma_{add}, M'\rangle$.
  By definition of $\longrightarrow^*$ *follows* $\langle \Gamma_{add}, M\rangle \xrightarrow{\sigma}{}^{*} \langle \Gamma_{add}, M'\rangle$.
  By the induction hypothesis follows $(\langle Sum(n-1), M'\rangle, \langle \Gamma_{add}, M'\rangle) \in \mathcal{R}$. □

The method of statebased simulation in principle suffices for proving any (valid) refinement. However, proving that relations are statebased simulations invites operational reasoning. As schedules get larger, this may become rather complex and therefore error-prone.

In Section 4.3.2 we have already shown that a compositional method of reasoning about refinement of schedules can not be based on strong statebased simulation. The same problems prohibit this for weak statebased simulation.

We conclude this section with a result which suggests a method other than simulation for establishing that a schedule describes a proper strategy for implementing a Gamma program.

Lemma 4.3.21 proves that if a schedule only consists of the rules of a given program and the outputs of the schedule are a subset of the outputs of that program, then that schedule is a weak statebased refinement of (the most general schedule of) that program.

Lemma 4.3.21 *Let P be simple program and s be a schedule.*
If $\mathcal{L}(s) \lhd \mathcal{L}(P)$ and $\mathcal{C}(s, M_0) \subseteq \mathcal{C}(P, M_0)$, then $\langle s, M_0\rangle \precapprox \langle \Gamma_P, M_0\rangle$.

Proof Let

$$\begin{aligned}\mathcal{R} \;=\; \{(\langle s, M\rangle, \langle t, M\rangle) \mid\; & \mathcal{L}(s) \lhd \mathcal{L}(P) \;\wedge\; \mathcal{C}(s, M) \subseteq \mathcal{C}(P, M_0) \wedge \\ & \langle t, M\rangle \text{ is } \langle \Gamma_P, M_0\rangle\text{-derived where } t \equiv t' \parallel \Gamma_P\}\end{aligned}$$

Clearly $(\langle s, M_0\rangle, \langle \Gamma_P, M_0\rangle) \in \mathcal{R}$. The result follows by showing that $\mathcal{R}$ is a weak statebased simulation.
transition
Suppose $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$. Then $\mathcal{L}(s') \subseteq \mathcal{L}(s)$ and, by Lemma 4.3.11, $\mathcal{C}(s', M') \subseteq \mathcal{C}(s, M)$. Then from $\mathcal{L}(s) \subseteq \mathcal{L}(P)$ follows $\mathcal{L}(s') \subseteq \mathcal{L}(P)$ and from $\mathcal{C}(s, M) \subseteq \mathcal{C}(P, M_0)$ follows $\mathcal{C}(s', M') \subseteq \mathcal{C}(P, M_0)$. Consider the cases $\lambda = \varepsilon$ and $\lambda = \sigma$.

- $\lambda = \varepsilon$: Then $M' = M$ and $\langle t, M\rangle \xrightarrow{\langle\,\rangle}{}^{*} \langle t, M\rangle$. Clearly $\langle t, M\rangle$ is $\langle \Gamma_P, M_0\rangle$-derived, hence $(\langle s', M'\rangle, \langle t, M'\rangle) \in \mathcal{R}$.

- $\lambda = \sigma$: Then, by Lemma 3.3.22 follows $\langle \Gamma_P, M\rangle \xrightarrow{\sigma} \langle t'', M'\rangle$ where $t'' \equiv t''' \parallel \Gamma_P$. Since $t \equiv \Gamma_P \parallel t'$, we derive by (N2) and the definition of $\longrightarrow^*$, $\langle t, M\rangle \xrightarrow{\sigma}{}^{*} \langle t', M'\rangle$ where $t' \equiv t''' \parallel \Gamma_P$. Clearly $\langle t', M'\rangle$ is $\langle \Gamma_P, M_0\rangle$-derived, hence $(\langle s', M'\rangle, \langle t', M'\rangle) \in \mathcal{R}$.

termination
If $s \equiv \mathsf{skip}$, then from $M\mathcal{C}(s, M) \subseteq \mathcal{C}(P, M)$ follows $\langle P, M\rangle\surd$. Then, because t is $\langle \Gamma_P, M_0\rangle$-derived, $\langle t, M\rangle \xrightarrow{\varepsilon} \langle \mathsf{skip}, M\rangle$. □

4.3.4 Soundness of Weak Statebased Refinement

The power of weak refinement is that it is insensitive to a differing number of ε-transitions. However, this has the undesirable consequence that weak refinement does not preserve total correctness. Weak refinement allows the introduction of an arbitrary number of ε-transitions. In particular, we may introduce an infinite number of ε-transitions which may turn a terminating schedule into a diverging one, thereby invalidating total correctness.

Example 4.3.22 *We use $fail = \overline{x} \to m \Leftarrow false$ to denote a rewrite rule that can only make ε-transitions. Let $F \mathrel{\widehat{=}} fail; F$. It is straightforward to prove that F is a weak refine-*

ment of skip*; i.e.* $\langle F, M\rangle \precapprox \langle \mathsf{skip}, M\rangle$ *for any* M*. However replacing* skip *by* F *introduces an infinite sequence of* ε *transitions. In terms of the capability function, we have, for all* M*,* $\mathcal{C}(F, M) = \{\bot\}$ *and* $\mathcal{C}(\mathsf{skip}, M) = \{M\}$*. Hence, while* $\langle F, M\rangle \precapprox \langle \mathsf{skip}, M\rangle$*, we also have* $\mathcal{C}(F, M) \not\subseteq \mathcal{C}(\mathsf{skip}, M)$ *(cf. Theorem 4.3.9).*

In [90] (pp. 147-149) Milner runs into a similar problem. We share his opinion that in a theory which relates behaviours which may differ arbitrarily with respect to the number of actions without effect, it is natural to allow this number to be infinite.

Hence, when using weak refinement, one should realize that this does not guarantee preservation of termination behaviour. However, weak refinement does preserve partial correctness; i.e. if the refining schedule does terminate, then the resulting state is a final state of the refined schedule.

Theorem 4.3.23 *Let* $\langle s, M\rangle$ *and* $\langle t, M\rangle$ *be configurations such that* $\langle s, M\rangle \precapprox \langle t, M\rangle$*. Then* $(\mathcal{C}(s, M) \setminus \{\bot\}) \subseteq \mathcal{C}(t, M)$*.*

Proof Let $M' \in (\mathcal{C}(s, M) \setminus \{\bot\})$. Hence $\langle s, M\rangle \xrightarrow{\overline{\lambda}}{}^{*} \langle \mathsf{skip}, M'\rangle$ for some $\overline{\lambda}$. We show by induction on the length, say n, of the transition sequence that $\langle t, M\rangle \xrightarrow{\overline{\mu}}{}^{*} \langle \mathsf{skip}, M'\rangle$ (where $\widehat{\lambda} = \widehat{\mu}$).

- $n = 0$: Then $s \equiv \mathsf{skip}$, $M' = M$ and $\lambda = \langle\,\rangle$. From $\langle s, M\rangle \precapprox \langle t, M\rangle$ follows $\langle t, M\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle \mathsf{skip}, M\rangle$ where $\widehat{\lambda'} = \langle\,\rangle$.
- $n > 0$: The sequence of transitions can be split into $\langle s, M\rangle \xrightarrow{\lambda'} \langle s', M''\rangle \xrightarrow{\overline{\lambda''}}{}^{*} \langle \mathsf{skip}, M'\rangle$ where $\overline{\lambda} = \lambda' \cdot \overline{\lambda''}$. From the initial transition and $\langle s, M\rangle \precapprox \langle t, M\rangle$ follows $\langle t, M\rangle \xrightarrow{\overline{\mu}}{}^{*} \langle t', M''\rangle$ such that $\overline{\mu} = \varepsilon^k \cdot \widehat{\lambda}$ for some $k \geq 0$ and $\langle s', M''\rangle \precapprox \langle t', M''\rangle$. Then, for the remainder of the transition sequence follows by the induction hypothesis that $\langle t', M''\rangle \xrightarrow{\overline{\mu'}}{}^{*} \langle \mathsf{skip}, M'\rangle$ such that $\widehat{\lambda''} = \widehat{\mu'}$. By concatenation follows $\langle t, M\rangle \xrightarrow{\overline{\mu \cdot \mu'}}{}^{*} \langle \mathsf{skip}, M'\rangle$.

From $\langle t, M\rangle \xrightarrow{\overline{\mu}}{}^{*} \langle \mathsf{skip}, M'\rangle$ follows $M' \in \mathcal{C}(t, M)$. □

4.4 Stateless Refinement

An important concern of any formal method should be that it has to be of practical use. The method of statebased simulation (presented in the preceding sections) in principle suffices for proving any (valid) refinement, but has the practical disadvantage that the

induced refinement relation is not a precongruence for schedules. This means that refinement cannot be applied in a modular fashion. Hence schedules need to be considered as a whole, which may result in complex proofs.

The statebased notion of refinement fails to be a precongruence because the interaction that occurs when a schedule is placed in a context may give rise to behaviour that was not considered for the individual schedules. A solution is to devise a notion of refinement that relates schedules for a wider range of behaviours.

The notion of refinement that we consider in this section requires the behaviours of schedules to match while the multiset on which they operate may be changed in a completely arbitrary way at any stage of the execution. These arbitrary changes to the multiset model the potential interactions that may occur when a schedule is put into some context (e.g. composed with some other schedule). The fact that the multiset may change arbitrarily reflects a "worst case" assumption about the context, but ensures that any behaviour that may arise through the interaction of a schedule and a context in which it is placed, are already taken into account when considering the refinement of the individual schedule.

As a consequence of taking all interaction into account, the problems with compositionality described in Section 4.3.2 do not occur. This enables us to show that the notion of refinement we develop in this section is a precongruence, hence allows a modular and algebraic approach to refinement.

With statebased simulation, the next transition (and hence next multiset) of a configuration depends on the schedule and the multiset of the current configuration. Because we will allow arbitrary interference in the notion of refinement that we develop in this section, the notion of "current multiset" is meaningless. Therefore, the notion of refinement that we will develop in this section is called *stateless* refinement.

In this section, we first develop the theory of strong stateless simulation and refinement. Next, we present a number of algebraic laws that follow from this variant of refinement. Later, we look at the weak stateless variants and the additional laws it induces.

For stateless simulation the next transition does not depend on the multiset of a "current configuration". Therefore the multiset-component is omitted from the simulation relation. This yields the following definition.

Definition 4.4.1 *A relation* $\mathcal{R} \subseteq \mathbb{S} \times \mathbb{S}$ *is a* strong stateless simulation *if,*
for all $(s,t) \in \mathcal{R}$*, for all* λ*, for all* $M \in \mathbb{M}$

1. $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle \Rightarrow \exists t' : \langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle$ *such that* $(s', t') \in \mathcal{R}$
2. $s \equiv \mathsf{skip} \Rightarrow t \equiv \mathsf{skip}$

We continue by presenting some basic properties of strong stateless simulations.

Proposition 4.4.2 *Assume that each* $\mathcal{R}_i$ *for* $i = 1, 2, \ldots$ *is a strong stateless simulation. Then the following relations are all strong stateless simulations:*

1. *the identity relation on schedules:* $Id_{\mathbb{S}} = \{(s, s) \mid s \in \mathbb{S}\}$
2. *the composition:* $\mathcal{R}_1\mathcal{R}_2$
3. *the union:* $\bigcup_{i\in I} \mathcal{R}_i$

Proof Postponed to Section 5.2. □

As before, we define strong stateless refinement, denoted $\leqslant$, as the largest strong stateless simulation relation. We consider a pair of schedules to be strong stateless *equivalent*, denoted $\simeq$, if the refinement relation holds in both directions.

Definition 4.4.3

1. $\leqslant = \bigcup\{\mathcal{R} \mid \mathcal{R}$ *is a strong stateless simulation* $\}$
2. $\simeq \; = \; \leqslant \cap \leqslant^{-1}$

Proposition 4.4.4

1. $\leqslant$ *is the largest strong stateless simulation.*
2. $\leqslant$ *is a partial order.*
3. $\simeq$ *is an equivalence relation.*

Proof Postponed to Section 5.2. □

We see that using stateless simulation, s is a refinement of t, if t can match the transitions by s, independent of the multiset. This relation cannot be invalidated by some (demonic) modification of the multiset by the context in which that schedule executes. This has the beneficial consequence that stateless refinement is a precongruence.

Proposition 4.4.5 $\leqslant$ *is a precongruence on* $\mathbb{S}$.

Proof In Section 5.2 □

In Section 5.2 we show that $\leqslant$ is the largest relation that satisfies Definition 4.4.1. To establish $s \leqslant t$ it suffices to prove that a relation $\mathcal{R}$, where $(s,t) \in \mathcal{R}$, is a strong stateless simulation relation.

In Definition 4.4.6 we define the *up-to*-generalization of strong stateless simulation. This definition facilitates proving that some relation is a strong stateless simulation because it allows us to make use of the fact that we have already proven other relations to be refinements.

Definition 4.4.6 *A binary relation $\mathcal{R} \subseteq \mathbb{S} \times \mathbb{S}$ is a* strong stateless simulation up-to $\leqslant$ *if $(s,t) \in \mathcal{R}$ implies, for all λ, for all $M \in \mathbb{M}$,*

1. $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle \Rightarrow \exists t' : \langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle \ \wedge \ (s', t') \in \ \leqslant \mathcal{R} \leqslant$
2. $s \equiv \mathsf{skip} \ \Rightarrow \ t \equiv \mathsf{skip}$

From Proposition 4.4.7 follows that in order to show $s \leqslant t$, it suffices to show that s and t are related by some strong stateless simulation up-to $\leqslant$.

Proposition 4.4.7 *If $\mathcal{R}$ is a strong stateless simulation up-to $\leqslant$, then $\mathcal{R} \subseteq \ \leqslant$.*

Proof Postponed to Section 5.2. □

4.4.1 Soundness of Strong Stateless Refinement

Strong stateless refinement of schedules preserves the relational ordering on the set of possible outputs; i.e. if we refine a schedule s by s', then s' will produce an output we were willing to accept from s.

Theorem 4.4.8 *If $s \leqslant t$, then $\forall M : \mathcal{C}(s,M) \subseteq \mathcal{C}(t,M)$.*

Proof First recall that $C(s,M) \neq \emptyset$ for all s, M. Let $x \in \mathcal{C}(s,M)$, we have to show that $x \in \mathcal{C}(t,M)$.
Consider the following cases:

- $x = \perp$:
  Hence $\langle s, M\rangle = \langle s_0, M_0\rangle$ and for all $i \geq 0$ there exists a λ_i such that $\langle s_i, M_i\rangle \xrightarrow{\lambda_i} \langle s_{i+1}, M_{i+1}\rangle$. By $s \leqslant t$ follows $\langle t, M\rangle = \langle t_0, M_0\rangle$ and for all $i \geq 0$ there exists a λ_i such that $\langle t_i, M_i\rangle \xrightarrow{\lambda_i} \langle t_{i+1}, M_{i+1}\rangle$. Hence $\perp \in \mathcal{C}(t,M)$.

- $x = M'$:
  Hence $\langle s, M\rangle \xrightarrow{\bar{\lambda}}{}^{*} \langle \mathsf{skip}, M'\rangle$. By $s \leqslant t$ and induction on the length of the transition sequence follows $\langle t, M\rangle \xrightarrow{\bar{\lambda}}{}^{*} \langle \mathsf{skip}, M'\rangle$, hence $M' \in \mathcal{C}(t, M)$.

□

4.4.2 Laws for Strong Stateless Refinement

The precongruence of strong stateless refinement entails that the (in)equations it induces may be used in any context, hence can be considered as refinement laws. In this section we present a number of the basic refinement laws. These laws give insight into the algebraic properties of refinement. Furthermore, the laws give rise to an algebraic style of reasoning about schedules.

First, we prove Lemma 4.4.9 which is used in the proofs of the subsequent lemmas. It states that if two terms are related by structural congruence, then their behaviour is strong stateless equivalent.

Lemma 4.4.9 *Let $s, t \in \mathbb{S}$. If $s \equiv t$ then $s \simeq t$.*

Proof By definition $s \simeq t$ iff $s \leqslant t$ and $t \leqslant s$.

- $s \equiv t \;\Rightarrow\; s \leqslant t$:
  transition
  If $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$ then, by (N8) and $s \equiv t$ follows $\langle t, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$.
  By reflexivity of $\leqslant$ holds $s' \leqslant s'$.
  termination
  If $s \equiv \mathsf{skip}$, then by transitivity of $\equiv$ follows $t \equiv \mathsf{skip}$.

- $s \equiv t \;\Rightarrow\; t \leqslant s$: The proof is analogous to the previous case.

□

Next, we present the laws grouped per operator. The schedules in these laws range over $\mathbb{S}$.

Laws for Rule Conditional Composition

The first law can be used to move a single rule-conditional out of a parallel composition such that it is scheduled for execution first. The second law is a special case of the first. The fact that sequential composition enforces a more determined ordering on the execution of schedules than parallel composition, has as a consequence that the law for ";" is a congruence, while the case for " $\|$ " is a refinement.

Lemma 4.4.10

1. $r \to (s_1 \,\|\, t)[s_2 \,\|\, t] \leqslant (r \to s_1[s_2]) \,\|\, t$
2. $r \to (s_1; t)[s_2; t] \simeq (r \to s_1[s_2]); t$

Proof

1. *transition*: There are two possible transitions:
   - If $\langle r \to (s_1 \,\|\, t)[s_2 \,\|\, t], M\rangle \xrightarrow{\sigma} \langle s_1 \,\|\, t, M'\rangle$ then, by (N1), $\langle (r \to s_1[s_2]) \,\|\, t, M\rangle \xrightarrow{\sigma} \langle s_1 \,\|\, t, M'\rangle$. By reflexivity, $s_1 \,\|\, t \leqslant s_1 \,\|\, t$.
   - If $\langle r \to (s_1 \,\|\, t)[s_2 \,\|\, t], M\rangle \xrightarrow{\varepsilon} \langle s_2 \,\|\, t, M\rangle$ then, by (N0), $\langle (r \to s_1[s_2]) \,\|\, t, M\rangle \xrightarrow{\varepsilon} \langle s_2 \,\|\, t, M\rangle$. By reflexivity, $s_2 \,\|\, t \leqslant s_2 \,\|\, t$.

   termination: There are no s_1, s_2, t_1 and t_2 such that $r \to (s_1 \,\|\, t)[s_2 \,\|\, t] \equiv \mathsf{skip}$, hence this case holds vacuously.

2. We prove the following cases.
   - $(r \to s_1[s_2]); t \leqslant r \to (s_1; t)[s_2; t]$:
     transition: There are two possible transitions:
     - $\langle (r \to s_1[s_2]); t, M\rangle \xrightarrow{\varepsilon} \langle s_2; t, M\rangle$, which is derived by (N0) from $\langle r \to s_1[s_2], M\rangle \xrightarrow{\varepsilon} \langle s_2, M\rangle$. Then by (N0) we derive $\langle r \to (s_1; t)[s_2; t], M\rangle \xrightarrow{\varepsilon} \langle s_2; t, M\rangle$. By reflexivity, $s_2; t \leqslant s_2; t$.
     - $\langle (r \to s_1[s_2]); t, M\rangle \xrightarrow{\sigma} \langle s_1; t, M'\rangle$, which is derived by (N1) from $\langle r \to s_1[s_2], M\rangle \xrightarrow{\sigma} \langle s_1, M'\rangle$. Then by (N1) we derive $\langle r \to (s_1; t)[s_2; t], M\rangle \xrightarrow{\sigma} \langle s_1; t, M'\rangle$. By reflexivity, $s_1; t \leqslant s_1; t$.

     termination: There are no s_1, s_2 and t such that $r \to s_1[s_2]; t \equiv \mathsf{skip}$, hence this case holds vacuously.

- $r \to (s_1; t)[s_2; t] \leqslant (r \to s_1[s_2]); t$: The proof is analogous to the previous case.

□

Laws for Sequential Composition

The laws from Lemma 4.4.11 show that ";" is a monoid with unit skip.

Lemma 4.4.11

1. $\mathsf{skip}; s \simeq s$
2. $s; \mathsf{skip} \simeq s$
3. $s_1; (s_2; s_3) \simeq (s_1; s_2); s_3$

Proof
Cases 1 and 3 follow from structural congruence and Lemma 4.4.9. We consider case 2. We have to prove $s; \mathsf{skip} \leqslant s$ and $s \leqslant s; \mathsf{skip}$. We give the details for the former; the proof of the latter is analogous.

Let $\mathcal{R} = \{(s; \mathsf{skip}, s) \mid s \in \mathbb{S}\}$. We show that $\mathcal{R}$ is a strong stateless simulation.
transition
If $s \not\equiv \mathsf{skip}$, then a transition for $s; \mathsf{skip}$ can be derived by (N5) from $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$. By definition of $\mathcal{R}$: $(s'; \mathsf{skip}, s') \in \mathcal{R}$.
termination
$s; \mathsf{skip} \equiv \mathsf{skip}$ only if $s \equiv \mathsf{skip}$. □

Laws for Parallel Composition

The laws for parallel composition follow from structural congruence and Lemma 4.4.9. They show that " $\|$ " is a commutative monoid with unit skip.

Lemma 4.4.12

1. $\mathsf{skip} \,\|\, s \simeq s$
2. $s_1 \,\|\, s_2 \simeq s_2 \,\|\, s_1$
3. $s_1 \,\|\, (s_2 \,\|\, s_3) \simeq (s_1 \,\|\, s_2) \,\|\, s_3$

Proof By structural congruence and Lemma 4.4.9. □

Distributivity Laws for Parallel and Sequential Composition

The next Lemma yields a general law for the distribution of sequential and parallel composition.

Lemma 4.4.13 $(s_1 \parallel s_3);(s_2 \parallel s_4) \leqslant (s_1;s_2) \parallel (s_3;s_4)$

Proof
Let $\mathcal{R} = \{((s_1 \parallel s_3);(s_2 \parallel s_4),(s_1;s_2) \parallel (s_3;s_4)) \mid s_1,s_2,s_3,s_4 \in \mathbb{S}\} \cup Id_{\mathbb{S}}$. We show that $\mathcal{R}$ is a strong stateless simulation. By Proposition 4.4.2(1) follows that $Id_{\mathbb{S}}$ is a strong stateless simulation. We consider the remaining case.
transition
We consider the possible transitions.

- By rule (N5) a transition can be derived from $\langle s_1 \parallel s_3, M\rangle \xrightarrow{\lambda} \langle s_1' \parallel s_3', M'\rangle$.
  This may in turn be derived in one of the following ways.

  1. By (N2) from $\langle s_1, M\rangle \xrightarrow{\lambda} \langle s_1', M'\rangle$, hence $s_3' \equiv s_3$.
     By (N5) we get $\langle s_1;s_2, M\rangle \xrightarrow{\lambda} \langle s_1';s_2, M'\rangle$.
     By (N2) we infer $\langle (s_1;s_2) \parallel (s_3;s_4), M\rangle \xrightarrow{\lambda} \langle (s_1';s_2) \parallel (s_3;s_4), M'\rangle$.
     And $((s_1' \parallel s_3);(s_2 \parallel s_4),(s_1';s_2) \parallel (s_3;s_4)) \in \mathcal{R}$.
  2. By (N2) from $\langle s_3, M\rangle \xrightarrow{\lambda} \langle s_3', M'\rangle$, hence $s_1' \equiv s_1$.
     The proof is analogous to the previous case.
  3. By (N3) from $\langle s_1, M\rangle \xrightarrow{\lambda} \langle s_1', M'\rangle$ and $\langle s_3, M\rangle \xrightarrow{\varepsilon} \langle s_3', M\rangle$.
     By (N5) we get for the former $\langle s_1;s_2, M\rangle \xrightarrow{\lambda} \langle s_1';s_2, M'\rangle$,
     and for the latter $\langle s_3;s_4, M\rangle \xrightarrow{\varepsilon} \langle s_3';s_4, M\rangle$.
     Then by (N3) we obtain $\langle (s_1;s_2) \parallel (s_3;s_4), M\rangle \xrightarrow{\lambda} \langle (s_1';s_2) \parallel (s_3';s_4), M'\rangle$.
     Clearly $((s_1' \parallel s_3');(s_2 \parallel s_4),(s_1';s_2) \parallel (s_3';s_4)) \in \mathcal{R}$.
  4. By (N3) from $\langle s_1, M\rangle \xrightarrow{\varepsilon} \langle s_1', M\rangle$ and $\langle s_3, M\rangle \xrightarrow{\lambda} \langle s_3', M'\rangle$.
     The proof is analogous to the previous case.
  5. By (N4) from $\langle s_1, M\rangle \xrightarrow{\sigma_1} \langle s_1', M_1\rangle$ and $\langle s_3, M\rangle \xrightarrow{\sigma_2} \langle s_3', M_2\rangle$ such that $M \models \sigma_1 \bowtie \sigma_2$. The proof is analogous to the previous case where use of (N3) should be replaced by use of (N4).

- By (N8) and (E1) from $(s_1 \parallel s_3) \equiv \mathsf{skip}$ (hence $s_1 \equiv \mathsf{skip}$ and $s_3 \equiv \mathsf{skip}$), and $\langle s_2 \parallel s_4, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$. From $(\mathsf{skip}; s_2) \parallel (\mathsf{skip}; s_4) \equiv s_2 \parallel s_4$ we get by (N8) and (E1) that $\langle (\mathsf{skip}; s_2) \parallel (\mathsf{skip}; s_4), M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$. Clearly $(s', s') \in Id_{\mathbb{S}} \subseteq \mathcal{R}$.

termination
If $(s_1 \parallel s_3); (s_2 \parallel s_4) \equiv \mathsf{skip}$ then $s_i \equiv \mathsf{skip}$ for all $i : 1 \leq i \leq 4$.
Then also $(s_1; s_2) \parallel (s_3; s_4) \equiv \mathsf{skip}$. □

The refinement of Lemma 4.4.13 is represented graphically in Figure 4.4 in conformance with the conventions of Figure 4.1 with the exception that here an arrow may denote a sequence of transitions.

refines

Figure 4.4: Refinement of Lemma 4.4.13

The schedule on the right hand side of Lemma 4.4.13 consists of two "threads" $s_1; s_2$ and $s_3; s_4$ that can proceed independently of each other. For example, the thread $s_1; s_2$ may terminate while the other thread is still executing s_3. In the schedule on the left hand side, the semi-colon forces the two threads to synchronize after termination of s_1 and s_3; i.e. before starting execution of either s_2 or s_4.

Corollary 4.4.14 shows some special cases of Lemma 4.4.13. Especially the first of these will turn out to be very useful.

Corollary 4.4.14

1. $s_1; s_2 \leqslant s_1 \parallel s_2$

2. $s_1; (s_2 \parallel s_3) \leqslant (s_1; s_2) \parallel s_3$

3. $(s_1 \parallel s_3); s_2 \leqslant (s_1; s_2) \parallel s_3$

Proof Take one or two terms of Lemma 4.4.13 equal to skip. Eliminate skip-terms using Lemma 4.4.11. □

Laws for Conditional Composition

In Lemma 4.4.15 we present some basic and distributive laws for the conditional combinators.

Lemma 4.4.15

1. $\mathit{false} \triangleright s[t] \simeq t$
2. $\mathit{true} \triangleright s[t] \simeq s$
3. $c \triangleright \mathsf{skip} \simeq \mathsf{skip}$
4. $c \triangleright (s_1 \parallel s_2)[t_1 \parallel t_2] \simeq (c \triangleright s_1[t_1]) \parallel (c \triangleright s_2[t_2])$
5. $c \triangleright (s_1; s_2)[t_1; t_2] \simeq (c \triangleright s_1[t_1]); (c \triangleright s_2[t_2])$
6. $!(c \triangleright s[t]) \simeq c \triangleright (!s)[!t]$

Proof By propositional calculus and structural congruence. □

The next laws may be used to eliminate or combine conditionals.

Lemma 4.4.16

1. $c \triangleright s[t] \simeq c \triangleright s\,[\neg c \triangleright t]$
2. $c \triangleright s[t] \simeq (c \triangleright s) \parallel (\neg c \triangleright t)$
3. $c \triangleright (r \rightarrow s[t]) \simeq c \triangleright (r \rightarrow c \triangleright s[t])$
4. $(c_1 \wedge c_2) \triangleright s[t] \simeq c_1 \triangleright (c_2 \triangleright s[t])[t]$

Proof By propositional calculus and structural congruence. □

Next, we investigate how the conditional $c \triangleright ..[..]$ can be combined with the rule conditional $r \rightarrow ..[..]$. Lemma 4.4.17 describes a refinement which is based on the idea

that the condition c can be used to test whether or not a rewrite rule r can be executed successfully We use *fail* to denote a rewrite rule that never succeeds (can only make ε-transitions). We can think of it as being defined as $\mathit{fail} \mathrel{\widehat{=}} \overline{x} \to m \Leftarrow \mathit{false}$. For any rule r holds $\mathit{fail} \lhd r$, hence *fail* is a lower bound for the set of multiset rewrite rules ordered by the strengthening relation $\lhd$.

In the following laws, we use $c \Rightarrow \neg b$ to mean: for all valuations $\overline{v}$, $c[\overline{x} := \overline{v}] \Rightarrow \neg b[\overline{x} := \overline{v}]$.

Lemma 4.4.17 *Let* $r = \overline{x} \mapsto m \Leftarrow b$. *If* $c \Rightarrow \neg b$, *then* $c \rhd (\mathit{fail}; s_2)[t] \simeq c \rhd (r \to s_1[s_2])[t]$.

Proof Consider the following cases

- $c = \mathit{false}$: then by structural congruence and Lemma 4.4.9 $c \rhd (\mathit{fail}; s_2)[t] \simeq t$ and $c \rhd (r \to s_1[s_2])[t] \simeq t$. By reflexivity $t \simeq t$.
- $c = \mathit{true}$: then by structural congruence and Lemma 4.4.9 $c \rhd (\mathit{fail}; s_2)[t] \simeq \mathit{fail}; s_2$ and $c \rhd (r \to s_1[s_2])[t] \simeq r \to s_1[s_2]$.
  For $\mathit{fail}; s_2$, we infer by (N0) and (N5), $\langle \mathit{fail}; s_2, M\rangle \xrightarrow{\varepsilon} \langle s_2, M\rangle$.
  From $c \Rightarrow \neg b$ follows by (N0), $\langle r \to s_1[s_2], M\rangle \xrightarrow{\varepsilon} \langle s_2, M\rangle$. By reflexivity $s_2 \simeq s_2$.

□

Corollary 4.4.18 $\mathit{fail}; t \simeq \mathit{fail} \to s[t]$

Proof Follows as a special case from Lemma 4.4.17 by taking $c = \mathit{true}$. □

Execution of *fail* never changes the input-output behaviour of a schedule (or program). Hence it can always be omitted. This could be formally justified if $\mathsf{skip} \leqslant \mathit{fail}$ would be a strong stateless refinement. However, this is not the case because the left hand side and the right hand side make a different number of transitions. Weak statebased refinement does not distinguish between differing numbers of ε-transition. In Section 4.4.3 we will develop the weak variant of stateless refinement and present the laws that it induces (which resolve the above issue).

Laws for Replication

In this section we will uncover the algebraic properties that characterize replication. The first two laws follow straightforwardly from the operational semantics.

Lemma 4.4.19 $s \leqslant !s$.

Proof
transition: Suppose $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$. Then by (N6) we infer $\langle !s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$.
By reflexivity of $\leqslant$ follows $s' \leqslant s'$.
termination: By (E8), $s \equiv \mathsf{skip}$ implies $!s \equiv \mathsf{skip}$. □

Lemma 4.4.20 $s \parallel !s \leqslant !s$

Proof
transition: Suppose $\langle s \parallel !s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$. Then by (N7) we infer $\langle !s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$.
By reflexivity of $\leqslant$ follows $s' \leqslant s'$.
termination: $s \parallel !s \equiv \mathsf{skip}$ only if $s \equiv \mathsf{skip}$, then by (E8) $!s \equiv \mathsf{skip}$. □

Recall that s^k stands for $k \geq 0$ copies of schedule s composed in parallel. Using the above we formally justify, by Corollary 4.4.21, the intuition that "$!s$" stands for an arbitrary number of copies of "s" composed in parallel.

Corollary 4.4.21 *For all* $k \geq 1 : s^k \leqslant !s$

Proof By induction on k.

- $k = 1$: By Lemma 4.4.19 follows $s \leqslant !s$.
- $k > 1$:

$$\begin{array}{lll} s^k & & \\ \simeq & & \text{definition } s^k \\ s \parallel s^{k-1} & & \\ \leqslant & & \text{induction hypothesis} \\ s \parallel !s & & \\ \leqslant & & \text{Lemma 4.4.20} \\ !s & & \end{array}$$

□

An important property of replication is its idempotence. As a stepping stone to the general result, we first prove the following simpler case.

Lemma 4.4.22 $!s \parallel !s \leqslant !s$

Proof

Let $\mathcal{R} = \{(t \parallel (!s \parallel !s), t \parallel !s) \mid s, t \in \mathbb{S}\} \cup Id_{\mathbb{S}}$. We prove that $\mathcal{R}$ is a strong stateless simulation by induction on the depth of the inference. We will use the following property of $\mathcal{R}$

$$\text{If } (s_1, s_2) \in \mathcal{R} \text{ and } t \in \mathbb{S}, \text{ then } (t \parallel s_1, t \parallel s_2) \in \mathcal{R} \qquad (*)$$

From Proposition 4.4.2.1 follows that $Id_{\mathbb{S}}$ is a strong stateless simulation. We consider the remaining case.

transition A transition for $t \parallel (!s \parallel !s)$ can be derived in the following ways

1. From (N2) by $\langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle$. Then by (N2) also $\langle t \parallel !s, M\rangle \xrightarrow{\lambda} \langle t' \parallel !s, M'\rangle$. Clearly $(t' \parallel (!s \parallel !s), t' \parallel !s) \in \mathcal{R}$.

2. From (N2) by $\langle !s \parallel !s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$. This transition can in turn be derived in five ways. Two of these are symmetric, hence we only need to consider three.

   (a) By (N2) from $\langle !s, M\rangle \xrightarrow{\lambda} \langle s'', M'\rangle$. This can be derived in two ways.

      i. By (N6) from $\langle s, M\rangle \xrightarrow{\lambda} \langle s'', M'\rangle$, hence $s' = s'' \parallel !s$. Then, by (N2), we derive $\langle s \parallel !s, M\rangle \xrightarrow{\lambda} \langle s'' \parallel !s, M'\rangle$. By (N7) we infer $\langle !s, M\rangle \xrightarrow{\lambda} \langle s'' \parallel !s, M'\rangle$. Hence by (N2) $\langle t \parallel !s, M\rangle \xrightarrow{\lambda} \langle t \parallel s'' \parallel !s, M'\rangle$. Clearly $(t \parallel s'' \parallel !s, t \parallel s'' \parallel !s) \in Id_{\mathbb{S}} \subseteq \mathcal{R}$.

      ii. By (N7) from $\langle s \parallel !s, M\rangle \xrightarrow{\lambda} \langle s'', M'\rangle$, hence $s' = s'' \parallel !s$. By (N2) we infer $\langle s \parallel !s \parallel !s, M\rangle \xrightarrow{\lambda} \langle s'' \parallel !s, M'\rangle$. The derivation for this transition is shorter than the derivation of the transition we want to prove the proposition for, hence by the induction hypothesis we get $\langle s \parallel !s, M\rangle \xrightarrow{\lambda} \langle s''', M'\rangle$ such that $(s'' \parallel !s, s''') \in \mathcal{R}$. By (N7) also $\langle !s, M\rangle \xrightarrow{\lambda} \langle s''', M'\rangle$. By (N2) $\langle t \parallel !s, M\rangle \xrightarrow{\lambda} \langle t \parallel s''', M'\rangle$. From $(s'' \parallel !s, s''') \in \mathcal{R}$ follows by (*) that $(t \parallel s'' \parallel !s, t \parallel s''') \in \mathcal{R}$.

   (b) By (N3) from $\langle !s, M\rangle \xrightarrow{\lambda} \langle s'', M'\rangle$ and $\langle !s, M\rangle \xrightarrow{\varepsilon} \langle s''', M\rangle$. The proof proceeds, analogously to the previous case, by induction on the depth of the inference (where (N3) is used in place of (N2)).

   (c) By (N4) from $\langle !s, M\rangle \xrightarrow{\sigma_1} \langle s_1, M_1\rangle$, and $\langle !s, M\rangle \xrightarrow{\sigma_2} \langle s_2, M_2\rangle$ where $M \models \sigma_1 \bowtie \sigma_2$. The proof is analogous to case (b) (where (N4) is used instead of (N3)).

3. By (N3) from $\langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle$ and $\langle !s \parallel !s, M\rangle \xrightarrow{\varepsilon} \langle s', M\rangle$. The proof is analogous to the case 2.

4. By (N3) from $\langle t, M\rangle \xrightarrow{\varepsilon} \langle t', M\rangle$ and $\langle !s \parallel !s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$. The proof is analogous to the case 2.

5. By (N4) from $\langle t, M\rangle \xrightarrow{\sigma_1} \langle t', M_1\rangle$ and $\langle !s \parallel !s, M\rangle \xrightarrow{\sigma_2} \langle s', M_2\rangle$ such that $M \models \sigma_1 \bowtie \sigma_2$. The proof is analogous to the case 2.

termination
$t \parallel !s \parallel !s \equiv \mathsf{skip}$ only if $t \equiv \mathsf{skip}$ and $!s \equiv \mathsf{skip}$, hence $t \parallel !s \equiv \mathsf{skip}$. □

Corollary 4.4.23

1. $\forall k : k \geq 1 : (!s)^k \leqslant !s$

2. $\forall k : k \geq 1 : s^k \parallel !s \leqslant !s$

Proof

1. By induction on k.

   - $k = 1$: By reflexivity of $\leqslant$ follows $!s \leqslant !s$.
   - $k > 1$:

$$\begin{array}{lll} & (!s)^k & \\ \simeq & & \text{definition } t^k \\ & !s \parallel (!s)^{k-1} & \\ \leqslant & & \text{induction hypothesis} \\ & !s \parallel !s & \\ \leqslant & & \text{Lemma 4.4.22} \\ & !s & \end{array}$$

2. Assume $k \geq 1$. We calculate as follows.

$$\begin{array}{lll} & s^k \parallel !s & \\ \leqslant & & \text{Lemma 4.4.19} \\ & (!s)^k \parallel !s & \\ \simeq & & \text{definition of } s^{k+1} \\ & (!s)^{k+1} & \\ \leqslant & & \text{case 1} \\ & !s & \end{array}$$

□

Finally we prove that replication is idempotent.

Lemma 4.4.24 $!(!s) \leqslant\!\!\cdot\; !s$

Proof
Let $\mathcal{R} = \{(t \parallel !(!s), t \parallel !s) \mid s, t \in \mathbb{S}\} \cup Id_{\mathbb{S}}$. We show that $\mathcal{R}$ is a strong stateless simulation up-to $\leqslant\!\!\cdot$. We will use the following property of $\mathcal{R}$:

$$\text{If } (s_1, s_2) \in \leqslant\!\!\cdot\mathcal{R}\leqslant\!\!\cdot \text{ and } t \in \mathbb{S}, \text{ then } (t \parallel s_1, t \parallel s_2) \in \leqslant\!\!\cdot\mathcal{R}\leqslant\!\!\cdot \quad (*)$$

From Proposition 4.4.2.1 follows that $Id_{\mathbb{S}}$ is a strong stateless simulation. By reflexivity of $\leqslant\!\!\cdot$ follows that $Id_{\mathbb{S}}$ is a strong stateless simulation up-to $\leqslant\!\!\cdot$. We consider the remaining case.
transition
We proceed by induction on the depth of the derivation of $\langle t \parallel !(!s), M\rangle \xrightarrow{\lambda} \langle t' \parallel s', M'\rangle$. This transition can be derived in the following ways.

1. By (N2) from $\langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle$. Then by (N2) $\langle t \parallel !s, M\rangle \xrightarrow{\lambda} \langle t' \parallel !s, M'\rangle$. Clearly $(t' \parallel !(!s), t' \parallel !s) \in \leqslant\!\!\cdot\mathcal{R}\leqslant\!\!\cdot$.

2. By (N2) from $\langle !(!s), M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$. This transition can be derived in two ways:

   (a) By (N6) from $\langle !s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$. Then by (N2) $\langle t \parallel !s, M\rangle \xrightarrow{\lambda} \langle t \parallel s', M'\rangle$. And $(t' \parallel s', t' \parallel s') \in Id_{\mathbb{S}} \subseteq \leqslant\!\!\cdot\mathcal{R}\leqslant\!\!\cdot$.

   (b) By (N7) from $\langle !s \parallel !(!s), M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$. By the induction hypothesis follows $\langle !s \parallel !s, M\rangle \xrightarrow{\lambda} \langle s'', M'\rangle$ such that $(s', s'') \in \leqslant\!\!\cdot\mathcal{R}\leqslant\!\!\cdot$. From Corollary 4.4.22 follows $\langle !s, M\rangle \xrightarrow{\lambda} \langle s''', M'\rangle$ such that $s'' \leqslant s'''$. By transitivity of $\leqslant\!\!\cdot$ follows that $(s', s''') \in \leqslant\!\!\cdot\mathcal{R}\leqslant\!\!\cdot$. By (N2) follows $\langle t \parallel !s, M\rangle \xrightarrow{\lambda} \langle t \parallel s''', M'\rangle$. From $(s', s''') \in \leqslant\!\!\cdot\mathcal{R}\leqslant\!\!\cdot$ and (*) follows $(t \parallel s', t \parallel s''') \in \leqslant\!\!\cdot\mathcal{R}\leqslant\!\!\cdot$.

3. By (N3) from $\langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle$ and $\langle !(!s), M\rangle \xrightarrow{\varepsilon} \langle s', M\rangle$. For the latter transition follows, analogous to case 2, that $\langle !s, M\rangle \xrightarrow{\varepsilon} \langle s'', M\rangle$ such that $(s', s'') \in \leqslant\!\!\cdot\mathcal{R}\leqslant\!\!\cdot$. From (N3) then follows $\langle t \parallel !s, M\rangle \xrightarrow{\lambda} \langle t' \parallel s'', M'\rangle$ and by (*) we conclude $(t' \parallel s', t' \parallel s'') \in \leqslant\!\!\cdot\mathcal{R}\leqslant\!\!\cdot$.

4. By (N3) from $\langle t, M\rangle \xrightarrow{\varepsilon} \langle t', M\rangle$ and $\langle !(!s), M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$.
   The proof is analogous to the case 3.

5. By (N4) from $\langle t, M\rangle \xrightarrow{\sigma_1} \langle t', M_1\rangle$ and $\langle !(!s), M\rangle \xrightarrow{\sigma_2} \langle s', M_2\rangle$ where $M \models \sigma_1 \bowtie \sigma_2$. The proof is analogous to the case 3.

termination
$t \parallel !(!s) \equiv \mathsf{skip}$ only if $t \equiv \mathsf{skip}$ and $!(!s) \equiv \mathsf{skip}$. From the latter follows by (E8) that $!s \equiv \mathsf{skip}$, hence $t \parallel !s \equiv \mathsf{skip}$. □

Lemma 4.4.25 $!(!s) \simeq !s$

Proof

- $!s \leqslant !(!s)$: follows from Lemma 4.4.19.
- $!(!s) \leqslant !s$: follows from Lemma 4.4.24.

□

The next lemma proves a refinement concerning distributivity of replication over parallel composition.

Lemma 4.4.26 $!(s_1 \parallel s_2) \leqslant (!s_1) \parallel (!s_2)$

Proof Let $\mathcal{R} = \{(t \parallel !(s_1 \parallel s_2), t \parallel (!s_1) \parallel (!s_2)) \mid t, s_1, s_2 \in \mathbb{S}\} \cup Id_{\mathbb{S}}$. We show that $\mathcal{R}$ is a strong stateless simulation up-to $\leqslant$. We will use that $\mathcal{R}$ satisfies the following property

$$\text{If } (s_1, s_2) \in \ \leqslant\mathcal{R}\leqslant \ \text{ and } t \in \mathbb{S}, \text{ then } (t \parallel s_1, t \parallel s_2) \in \ \leqslant\mathcal{R}\leqslant \quad (*)$$

From Proposition 4.4.2.1 follows that $Id_{\mathbb{S}}$ is a strong stateless simulation. By reflexivity of $\leqslant$ follows that $Id_{\mathbb{S}}$ is a strong stateless simulation up-to $\leqslant$. We consider the remaining case.
transition
By induction on the depth of the inference.

1. By (N2) from $\langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle$. Then by (N2) $\langle t \parallel !s_1 \parallel !s_2, M\rangle \xrightarrow{\lambda} \langle t' \parallel !s_1 \parallel !s_2, M'\rangle$. Clearly $(t' \parallel !(s_1 \parallel s_2), t' \parallel (!s_1) \parallel (!s_2) \in \ \leqslant\mathcal{R}\leqslant$.
2. By (N2) from $\langle !(s_1 \parallel s_2), M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$. This transition can be derived in 2 ways.
   (a) by (N6) from $\langle s_1 \parallel s_2, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$. Transitions for $s_1 \parallel s_2$ can be derived in five ways. By symmetry of " $\parallel$ " we only have to consider three cases.

i. By (N2) from $\langle s_1, M\rangle \xrightarrow{\lambda} \langle s_1', M'\rangle$ hence $\langle !(s_1 \parallel s_2), M\rangle \xrightarrow{\lambda} \langle s_1' \parallel s_2, M'\rangle$. By (N6) we infer from the former $\langle !s_1, M\rangle \xrightarrow{\lambda} \langle s_1', M'\rangle$. By (N2) we obtain $\langle t \parallel !s_1 \parallel !s_2, M\rangle \xrightarrow{\lambda} \langle t \parallel s_1' \parallel !s_2, M'\rangle$. Because $s_2 \leqslant !s_2$ and $(t \parallel s_1' \parallel s_2, t \parallel s_1' \parallel s_2) \in \mathcal{R}$ we have $(t \parallel s_1' \parallel s_2, t \parallel s_1' \parallel !s_2) \in \ \leqslant\mathcal{R}\leqslant$.

ii. By (N3) from transitions $\langle s_1, M\rangle \xrightarrow{\lambda} \langle s_1', M'\rangle$ and $\langle s_2, M\rangle \xrightarrow{\varepsilon} \langle s_2', M\rangle$, hence $\langle !(s_1 \parallel s_2), M\rangle \xrightarrow{\lambda} \langle s_1' \parallel s_2', M'\rangle$. By (N6) we infer $\langle !s_1, M\rangle \xrightarrow{\lambda} \langle s_1', M'\rangle$ and $\langle !s_2, M\rangle \xrightarrow{\varepsilon} \langle s_2', M\rangle$. By (N3) we get $\langle !s_1 \parallel !s_2, M\rangle \xrightarrow{\lambda} \langle s_1' \parallel s_2', M'\rangle$. By (N2) we obtain $\langle t \parallel !s_1 \parallel !s_2, M\rangle \xrightarrow{\lambda} \langle t \parallel s_1' \parallel s_2', M'\rangle$. By reflexivity of $\leqslant$ and $Id_{\mathbb{S}} \subseteq \mathcal{R}$ follows $(t \parallel s_1' \parallel s_2', t \parallel s_1' \parallel s_2') \in \ \leqslant\mathcal{R}\leqslant$.

iii. By (N4) from $\langle s_1, M\rangle \xrightarrow{\sigma_1} \langle s_1', M_1\rangle$ and $\langle s_2, M\rangle \xrightarrow{\sigma_2} \langle s_2', M_2\rangle$ with $M \models \sigma_1 \bowtie \sigma_2$. The proof proceeds analogously to the preceding case.

(b) by (N7) from $\langle (s_1 \parallel s_2) \parallel !(s_1 \parallel s_2), M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$. By the induction hypothesis we get $\langle (s_1 \parallel s_2) \parallel (!s_1) \parallel (!s_2), M\rangle \xrightarrow{\lambda} \langle s'', M'\rangle$ such that $(s', s'') \in \ \leqslant\mathcal{R}\leqslant$. By Lemma 4.4.12(3) this can be equivalently written as $\langle (s_1 \parallel !s_1) \parallel (s_2 \parallel !s_2), M\rangle \xrightarrow{\lambda} \langle s'', M'\rangle$. From Lemma 4.4.20 and Proposition 4.4.5 follows $s_1 \parallel !s_1 \parallel s_2 \parallel !s_2 \leqslant !s_1 \parallel !s_2$, hence $\langle (!s_1) \parallel (!s_2), M\rangle \xrightarrow{\lambda} \langle s''', M'\rangle$ such that $s'' \leqslant s'''$. By (N2) we infer $\langle t \parallel (!s_1) \parallel (!s_2), M\rangle \xrightarrow{\lambda} \langle t \parallel s''', M'\rangle$. From $(s', s'') \in \ \leqslant\mathcal{R}\leqslant$ and $(s'', s''') \in \ \leqslant$ we get by transitivity of $\leqslant$ that $(s', s''') \in \ \leqslant\mathcal{R}\leqslant$, hence by (*) follows $(t \parallel s', t \parallel s''') \in \ \leqslant\mathcal{R}\leqslant$.

3. by (N3) from $\langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle$ and $\langle !(s_1 \parallel s_2), M\rangle \xrightarrow{\varepsilon} \langle s', M\rangle$.
The proof is a routine combination of the preceding cases.

4. by (N3) from $\langle t, M\rangle \xrightarrow{\varepsilon} \langle t', M\rangle$ and $\langle !(s_1 \parallel s_2), M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$.
The proof is a routine combination of the preceding cases.

5. by (N4) from $\langle t, M\rangle \xrightarrow{\sigma_1} \langle t', M_1\rangle$ and $\langle !(s_1 \parallel s_2), M\rangle \xrightarrow{\sigma_1} \langle s', M_2\rangle$ where $M \models \sigma_1 \bowtie \sigma_2$.
The proof is analogous to the preceding case.

termination
$t \parallel !(s_1 \parallel s_2) \equiv \mathsf{skip}$ implies $t \equiv \mathsf{skip}$ and $s_1 \equiv \mathsf{skip}$ and $s_2 \equiv \mathsf{skip}$. Then also $t \parallel (!s_1) \parallel (!s_2) \equiv \mathsf{skip}$. □

An interesting consequence from the idempotence of replication is that if a single instance of a schedule is a strong stateless refinement of the most general schedule, then

the replication of that schedule is also a strong stateless refinement of the most general schedule.

Lemma 4.4.27 *Let P be a simple program. If $s \leqslant \Gamma_P$, then $!s \leqslant \Gamma_P$.*

Proof

$$\begin{array}{lll} & !s & \\ \leqslant & & s \leqslant \Gamma_P, \text{ Proposition 4.4.5} \\ & !\Gamma_P & \\ \simeq & & \text{definition of } \Gamma_P \\ & !!\Pi_P & \\ \simeq & & \text{Lemma 4.4.25} \\ & !\Pi_P & \\ \simeq & & \text{definition of } \Gamma_P \\ & \Gamma_P & \end{array}$$

□

Lemma 4.4.27 does not generalize to programs that are not simple because $!(s_1; s_2) \not\leqslant (!s_1); (!s_2)$.

We end this section by returning to the refinement of Example 4.3.7. There, simulation was used to prove the validity of the refinement. Here we will use the refinement laws. The example shows that the same refinement can be proven much more concisely using equational reasoning.

Example 4.4.28 Let r_1 and r_2 be rules, then

$$(r_1; r_2) \parallel (r_2; r_1) \leqslant !(r_1 \parallel r_2)$$

Comparing the algebraic proof below with the proof by simulation of Example 4.3.7 illustrates that the former is a more convenient proof technique.

Proof

$$\begin{array}{lll} & (r_1; r_2) \parallel (r_2; r_1) & \\ \leqslant & & \text{Corollary 4.4.14.1} \\ & (r_1 \parallel r_2) \parallel (r_2 \parallel r_1) & \\ \simeq & & \text{Lemma 4.4.12.2} \\ & (r_1 \parallel r_2) \parallel (r_1 \parallel r_2) & \\ \leqslant & & \text{Corollary 4.4.21} \\ & !(r_1 \parallel r_2) & \end{array}$$

□

4.4.3 Weak Stateless Refinement

In Section 4.3.3 we exploited that failing rewrites (corresponding to ε-labelled transitions) are irrelevant to the outcome of a computation. We shall do the same here by developing the weak variant of stateless simulation, which is indifferent to ε-transitions.

Definition 4.4.29 *A relation $\mathcal{R} \subseteq \mathbb{S} \times \mathbb{S}$ is a* weak stateless simulation *if, for all $(s,t) \in \mathcal{R}$, for all λ, for all $M \in \mathbb{M}$*

1. $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle \Rightarrow \exists t' : \langle t, M\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle t', M'\rangle$ *such that* $(s', t') \in \mathcal{R}$ *and* $\overline{\lambda'} = \varepsilon^k \cdot \widehat{\lambda}$ *for some* $k \geq 0$
2. $s \equiv \mathsf{skip} \;\Rightarrow \langle t, M\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle \mathsf{skip}, M\rangle$ *where* $\widehat{\lambda'} = \langle\ \rangle$

We say that s is a *weak stateless refinement* of t, denoted $s \precsim t$, if $(s,t) \in \mathcal{R}$ for some weak stateless simulation $\mathcal{R}$. Schedules s and t are weak stateless equivalent, denoted $s \sim t$, if s is a weak stateless refinement of t and vice versa.

Definition 4.4.30

1. $\precsim \;= \bigcup\{\mathcal{R} \mid \mathcal{R}$ *is a weak stateless simulation* $\}$
2. $\sim \;=\; \precsim \cap \precsim^{-1}$

Lemma 4.4.31

1. $\precsim$ *is the largest weak stateless simulation.*
2. $\precsim$ *is a partial order.*
3. $\sim$ *is an equivalence relation.*

Proof Postponed to Section 5.5. □

The *up-to* technique (from [90]) can be adapted straightforwardly to weak stateless refinement.

Definition 4.4.32 *A relation $\mathcal{R} \subseteq \mathbb{S} \times \mathbb{S}$ is a* weak stateless simulation up-to $\precsim$ *if, for all $(s,t) \in \mathcal{R}$, for all M,*

1. $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle \Rightarrow \exists t' : \langle t, M\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle t', M'\rangle$ *such that* $s' \precsim\!\!\approx \mathcal{R} \approx\!\!\precsim t'$ *and* $\overline{\lambda'} = \varepsilon^k \cdot \widehat{\lambda}$ *for some* $k \geq 0$
2. $s \equiv \mathsf{skip} \Rightarrow \langle t, M\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle \mathsf{skip}, M\rangle$ *where* $\widehat{\lambda'} = \langle\ \rangle$

Lemma 4.4.33 *If* $\mathcal{R}$ *is a weak stateless simulation up-to* $\precsim$ *refinement, then* $\mathcal{R} \subseteq \precsim$.

Proof Postponed to Section 5.5. □

As was the case for strong stateless refinement, weak stateless refinement is a precongruence.

Proposition 4.4.34 $\precsim$ *is a precongruence for* $\mathbb{S}$.

Proof Postponed to Section 5.5. □

The essential difference between weak and strong stateless refinement can be expressed as an algebraic law. In this law we use the rewrite rule *fail* that represents a rule that can only make a failing transition (labelled by ε). We then arrive at the following weak stateless law, which enables the elimination of *fail* from schedules.

Lemma 4.4.35 $t \sim \mathit{fail} \rightarrow s[t]$

Proof

- $t \precsim \mathit{fail} \rightarrow s[t]$: Let $\mathcal{R} = \{(t, \mathit{fail} \rightarrow s[t]) \mid s, t \in \mathbb{S}\} \cup Id_{\mathbb{S}}$.
  We show that $\mathcal{R}$ is a weak stateless simulation.
  transition
  If $\langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle$, then, by $(N0)$, $\langle \mathit{fail} \rightarrow s[t], M\rangle \xrightarrow{\varepsilon} \langle t, M\rangle$.
  By transitivity of $\longrightarrow^*$ follows $\langle \mathit{fail} \rightarrow s[t], M\rangle \xrightarrow{\langle \varepsilon \cdot \lambda\rangle}{}^{*} \langle t', M'\rangle$.
  If $\lambda = \varepsilon$, then $\langle\, \varepsilon \cdot \lambda\,\rangle = \varepsilon^k \cdot \widehat{\lambda}$ for $k = 2$. Otherwise, if $\lambda = \sigma$, then $\widehat{\lambda} = \sigma$, hence $\langle\, \varepsilon \cdot \lambda\,\rangle = \varepsilon^k \cdot \widehat{\lambda}$ for $k = 1$. Clearly $(t', t') \in Id_{\mathbb{S}} \subseteq \mathcal{R}$.
  termination
  If $t \equiv \mathsf{skip}$, then by $(N0)$ and definition of $\longrightarrow^*$, $\langle \mathit{fail} \rightarrow s[t], M\rangle \xrightarrow{\varepsilon}{}^{*} \langle t, M\rangle$.
  Clearly $\widehat{\varepsilon} = \langle\ \rangle$.

- $\mathit{fail} \rightarrow s[t] \precsim t$: Let $\mathcal{R} = \{(\mathit{fail} \rightarrow s[t], t) \mid s, t \in \mathbb{S}\} \cup Id_{\mathbb{S}}$.
  We show that $\mathcal{R}$ is a weak stateless simulation.
  transition

By $(N0)$, $\langle \mathit{fail} \to s[t], M\rangle \xrightarrow{\varepsilon} \langle t, M\rangle$. By reflexivity of $\longrightarrow^*$: $\langle t, M\rangle \xrightarrow{\langle\,\rangle}{}^* \langle t, M\rangle$.
Clearly $\langle\,\rangle = \varepsilon^k \cdot \widehat{\varepsilon}$ for $k = 0$ and $(t, t) \in Id_{\mathbb{S}} \subseteq \mathcal{R}$.
termination
There are no s and t such that $\mathit{fail} \to s[t] \equiv \mathsf{skip}$, hence this case holds vacuously.

□

Corollary 4.4.36 $\mathsf{skip} \sim \mathit{fail}$

Proof From Lemma 4.4.35 by taking both $s \equiv \mathsf{skip}$ and $t \equiv \mathsf{skip}$. □

These laws may seem futile because no sensible program or schedule uses *fail*. However, consider the following (strong stateless) law that relates the schedule conditional $c \triangleright \,..\,[\,..\,]$ to the reaction condition b of a rewrite rule $\overline{x} \mapsto m \Leftarrow b$. Lemma 4.4.37 is an equivalence that introduces *fail*.

Lemma 4.4.37 *Let* $r = \overline{x} \mapsto m \Leftarrow b$. *If* $b \Rightarrow c$ *then* $r \to s[t] \simeq c \triangleright (r \to s[t])[\mathit{fail}; t]$.

Proof If $c = \mathit{true}$, then by structural congruence $c \triangleright (r \to s[t])[\mathit{fail}; t] \equiv r \to s[t]$, and the result follows by reflexivity of $\simeq$. It remains to consider the case $c = \mathit{false}$. From contrapositive of $b \Rightarrow c$ follows $\neg b$. From structural congruence follows $c \triangleright (r \to s[t])[\mathit{fail}; t] \equiv \mathit{fail}; t$.

- We show that $\mathcal{R} = \{(r \to s[t], \mathit{fail}; t) \mid \neg b, s, t \in \mathbb{S}\} \cup Id_{\mathbb{S}}$ is a strong stateless simulation.
  By reflexivity of $\leqslant$ follows that $Id_{\mathbb{S}}$ is a strong stateless simulation. We concentrate on the remaining terms. We consider the possible transitions.
  transition
  From $\neg b$, we get by $(N0)$, $\langle r \to s[t], M\rangle \xrightarrow{\varepsilon} \langle t, M\rangle$.
  By $(N0)$ and $(N5)$, $\langle \mathit{fail}; t, M\rangle \xrightarrow{\varepsilon} \langle t, M\rangle$, and $(t, t) \in Id_{\mathbb{S}} \subseteq \mathcal{R}$.
  termination
  There are no s and t such that $r \to s[t] \equiv \mathsf{skip}$, hence this case holds vacuously.

- We show that $\mathcal{R} = \{(\mathit{fail}; t, r \to s[t]) \mid \neg b, s, t \in \mathbb{S}\} \cup Id_{\mathbb{S}}$ is a strong stateless simulation.
  By reflexivity of $\leqslant$ we know that $Id_{\mathbb{S}}$ is a strong stateless simulation. We concentrate on the remaining terms. We consider the possible transitions.

transition
By $(N0)$, $\langle \mathit{fail}; t, M\rangle \xrightarrow{\varepsilon} \langle t, M\rangle$.
Because $\neg b$, we get by $(N0)$, $\langle r \to s[t], M\rangle \xrightarrow{\varepsilon} \langle t, M\rangle$, and $(t,t) \in Id_{\mathbb{S}} \subseteq \mathcal{R}$.
termination
There are no s and t such that $\mathit{fail}; t \equiv \mathsf{skip}$, hence this case holds vacuously.

□

A rewrite rule that leads to an ε-transition indicates that there is no data in the multiset which enables this rewrite rule. This conclusion can only be drawn after all possible combinations of all data in the multiset have been considered. This is generally a computationally intensive process. Hence, eventhough scheduling a failing rewrite rule does not affect the outcome of a schedule, it does affect the computational effor that takes place. From this perspective it is desirable to eliminate failing rewrite rules from schedules as much as possible.

In Chapter 5, we present a generic theory of refinement which justifies that the notions of refinement that we have seen may be applied in combination. In particular we may combine strong and weak stateless laws to derive weak stateless refinements. For example, by combining the weak stateless law of Lemma 4.4.35 and the strong stateless of Lemma 4.4.37 we derive the following weak stateless equivalence

Corollary 4.4.38 *Let* $r = \overline{x} \mapsto m \Leftarrow b$. *If* $b \Rightarrow c$, *then* $r \to s[t] \sim c \triangleright (r \to s[t])[t]$

Proof From Lemmas 4.4.35 and 4.4.37. □

4.5 Concluding Remarks

In this chapter we used the concept of simulation to develop formal notions for the refinement of coordination strategies. The main variants of refinement that we considered are statebased and stateless refinement.

The statebased notion assesses whether the behaviour of a refined schedule may evolve identically to that of the refining schedule for a particular multiset. This assumes that, throughout execution of these schedules, the multiset is not changed by an environment in which these schedules might operate. In contrast, the stateless notion considers whether

the refined schedule may display the same behaviour as the refining schedule while the multiset may be changed at any stage during execution.

These statebased and stateless variants have different strengths and weaknesses. The statebased variant is a powerful notion in the sense that it is, in principle, adequate for proving any refinement that we would expect to be valid based on the structural operational semantics. With statebased refinement it is possible to use properties of the multiset to justify refinements. However, statebased refinement is not a precongruence. Hence for proving a refinement, schedules need to be considered as a whole.

In contrast, stateless refinement is a precongruence. As a result, stateless refinement induces a number of interesting laws which can be used to reason about refinement of schedules in a modular, equational style. This method of reasoning is often more practical than the method of simulations required by statebased refinement. However, the price to be paid is that stateless refinement is less powerful. More precisely, stateless refinement fails to justify refinements that depend on properties of the multiset.

For both the statebased and stateless notion of refinement, we defined a strong and a weak variant. The strong variants require that every single transition of the refining schedule is matched by a single transition of the refined schedule. The weak variant relaxes this property by allowing the refining schedule and the refined schedule to differ with respect to the number of transitions that do not affect the multiset.

The notion of stateless refinement and its precongruence are first described in [26] and published as [30]. The statebased notion and the weak variants of stateless and statebased refinement were first described in [27] and published as [29].

5 A Generic Theory of Refinement

In the previous chapter we argued that it is desirable for a notion of refinement to be a precongruence, because this entails that the (in)equations it induces may be used in a modular, algebraic manner. However, we also observed that the precongruent notion stateless refinement does not justify as many refinements as the statebased notion which is not a precongruence. Hence, there is a trade-off between the ease of application and the scope of a notion of refinement.

In this chapter we develop a theory which is aimed at understanding the prerequisites for precongruence and the degree to which these influence the scope of application.

5.1 Introduction

A requirement for modular replacement of a schedule by a refining schedule is that the refinement relation is a precongruence; i.e. the refinement relation between these schedules must hold in any context[1] in which they may occur. Hence, a notion of refinement for schedules can only be a precongruence, if it takes into account, for a schedule and its refinement, all possible behaviours that may arise under the assumption that some context is also modifying the multiset. Because a modification of the multiset by the context may influence the behaviour of a schedule under consideration in an undesirable way, we call such a modification an *interference*.

The main difference between statebased and stateless refinement are their assumptions about the possible interference from the context. Technically, this interference is reflected in the (set of) multiset(s) that are considered as point of departure for the next transition.

Statebased refinement considers only transitions that depart from the configuration

[1] We will interchangeably use "context" and "environment" to denote the schedule that the schedule that we want to refine is part of. For example, the context of s in $s \parallel t$ is t.

that was arrived at by the previous transition. Hence, this notion does not take into account transitions that depart from configurations which may be arrived at by rewrites performed by the context in which the schedules may be executing. This is adequate if there are no schedules running in parallel (i.e. the schedule is considered as a whole) or if the schedules that form the context of the schedule under study do not interfere with the multiset.

Stateless refinement on the other hand considers, for every transition, departing configurations where the multiset is arbitrary; i.e. all possible multisets are considered. Hence, every transition may depart from a configuration where the multiset may differ in a completely arbitrary way from the multiset of the configuration arrived at by the previous transition. This can be interpreted as reflecting the possibility of an arbitrary interference. This assumption about the context is typical for so-called "open systems" where nothing is known about the environment.

We can observe a trade-off between the ease of use (precongruence) and the power of a refinement notion (how many refinements are justified) depending on what assumptions are made about the possible interferences from the environment. This raises the following question: What are suitable assumptions about the environment such that it is possible to use properties of the multiset, while the corresponding refinement relation is a precongruence?

This question is answered by developing a generic theory of refinement which is parameterized by the possible interferences. We can choose the interference parameter to capture assumptions about the environment.

This theory provides a unifying framework for simulation-based approaches for refinement of our coordination language. We show that the notions of refinement studied in Chapter 4 can be obtained as specific instances.

Furthermore, this generic theory reveals under which conditions on the interference-parameter the corresponding refinement relation enjoys desirable properties. An important property that can be predicted by this theory is whether a particular choice for the interference parameter yields a precongruent notion of refinement.

In the following sections we will subsequently develop the theory of strong and weak generic notion of refinement. Based on these results we present, in Chapter 6, a new precongruent notion of refinement and show that this gives rise to additional refinement laws.

5.2 Strong Generic Refinement

In this section we develop a generic theory of strong refinement by parameterizing the definition of simulation by a measure of interference.

We model interference by a relation $\phi \subseteq \mathbb{M} \times \mathbb{M}$, called the *interference set*, over pairs of multisets[2]. We use $(M, M') \in \phi$ to denote that M' is a multiset that may result from interference from the environment in a configuration with multiset M. Hence, if the current multiset is M, then the set of multisets in which we may end up in through interference from the environment is given by

$$\{M' \mid (M, M') \in \phi\}$$

Definition 5.2.1 shows how the interference parameter can be incorporated in the notion of simulation. According to this definition, one configuration is a refinement of another, if the configuration that is being refined is able to simulate all configurations that may result from interference with the current configuration.

Definition 5.2.1 *Let $\mathcal{R} \subseteq \mathbb{C} \times \mathbb{C}$ and $\phi \subseteq \mathbb{M} \times \mathbb{M}$.*
We say that $\mathcal{R}$ is a strong ϕ-simulation if for all $(\langle s, M\rangle, \langle t, N\rangle) \in \mathcal{R}$, for all λ, for all $(M, M') \in \phi$,

1. $M = N$
2. $\langle s, M'\rangle \xrightarrow{\lambda} \langle s', M''\rangle \Rightarrow \langle t, M'\rangle \xrightarrow{\lambda} \langle t', M''\rangle$ *and* $(\langle s', M''\rangle, \langle t', M''\rangle) \in \mathcal{R}$
3. $s \equiv \mathsf{skip} \Rightarrow t \equiv \mathsf{skip}$

We first prove some standard properties of ϕ-simulation.

Lemma 5.2.2 *If $\mathcal{R}_i$ are strong ϕ-simulations, then so are*

1. *the identity relation over configuration: $Id_{\mathbb{C}}$*
2. *the composition: $\mathcal{R}_1\mathcal{R}_2$*
3. *the union: $\bigcup_{i\in I} \mathcal{R}_i$*

[2]We interchangeably view ϕ as a relation or as a predicate over pairs of multisets by appealing to the correspondence $\phi(M, M') \Leftrightarrow (M, M') \in \phi$.

Proof

1. By reflexivity of $=$ and $\Rightarrow$.

2. Suppose $(\langle s_1, M\rangle, \langle s_2, M'\rangle) \in \mathcal{R}_1\mathcal{R}_2$, then for some t and N we have $(\langle s_1, M\rangle, \langle t, N\rangle) \in \mathcal{R}_1$ and $(\langle t, N\rangle, \langle s_2, M'\rangle) \in \mathcal{R}_2$. Because $\mathcal{R}_1$ and $\mathcal{R}_2$ are ϕ-simulations, we have $M = N = M'$.
   transition
   Now let $\phi(M, M')$ and $\langle s_1, M'\rangle \xrightarrow{\lambda} \langle s_1', M''\rangle$.
   Because $(\langle s_1, M\rangle, \langle t, M\rangle) \in \mathcal{R}_1$, there is some t' such that $\langle t, M'\rangle \xrightarrow{\lambda} \langle t', M''\rangle$ and $(\langle s_1', M''\rangle, \langle t', M''\rangle) \in \mathcal{R}_1$.
   Because $(\langle t, M\rangle, \langle s_2, M\rangle) \in \mathcal{R}_2$, there is some s_2' such that $\langle s_2, M'\rangle \xrightarrow{\lambda} \langle s_2', M''\rangle$ and $(\langle t', M''\rangle, \langle s_2', M''\rangle) \in \mathcal{R}_2$.
   From $(\langle s_1', M''\rangle, \langle t', M''\rangle) \in \mathcal{R}_1$ and $(\langle t', M''\rangle, \langle s_2', M''\rangle) \in \mathcal{R}_2$ follows $(\langle s_1', M''\rangle, \langle s_2', M''\rangle) \in \mathcal{R}_1\mathcal{R}_2$.
   termination
   If $s_1 \equiv \mathsf{skip}$ then from $(\langle s_1, M\rangle, \langle t, M\rangle) \in \mathcal{R}_1$ we have $t \equiv \mathsf{skip}$.
   From $(\langle t, M\rangle, \langle s_2, M\rangle) \in \mathcal{R}_2$ follows $s_2 \equiv \mathsf{skip}$.

3. Let $\mathcal{R} = \bigcup_{i\in I} \mathcal{R}_i$. Suppose $(\langle s_1, M\rangle, \langle s_2, N\rangle) \in \mathcal{R}$, then $(\langle s_1, M\rangle, \langle s_2, N\rangle) \in \mathcal{R}_i$ for some $i \in I$ hence $M = N$.
   transition
   If $\phi(M, M')$ and $\langle s_1, M'\rangle \xrightarrow{\lambda} \langle s_1', M''\rangle$ then, because $\mathcal{R}_i$ is a ϕ-simulation, we have $\langle s_2, M'\rangle \xrightarrow{\lambda} \langle s_2', M''\rangle$ and $(\langle s_1', M''\rangle, \langle s_2', M''\rangle) \in \mathcal{R}_i$.
   Because $\mathcal{R}_i \subseteq \mathcal{R}$ also $(\langle s_1', M''\rangle, \langle s_2', M''\rangle) \in \mathcal{R}$.
   termination
   The case $s_1 \equiv \mathsf{skip}$ goes analogously.

□

Next, we define strong ϕ-refinement, denoted $\leq^\phi$, as the maximal strong ϕ-simulation relation. Let $\langle s, M\rangle$ and $\langle t, N\rangle$ be configurations. We say that $\langle s, M\rangle$ is a strong ϕ-refinement of $\langle t, N\rangle$, denoted $\langle s, M\rangle \leq^\phi \langle t, N\rangle$, if $(\langle s, M\rangle, \langle t, N\rangle) \in \mathcal{R}$ for some strong ϕ-simulation $\mathcal{R}$. Strong ϕ-equivalence, denoted $=^\phi$, is defined as the intersection of strong ϕ-refinement and its inverse. We obtain a (family of) refinement relation(s) over schedules (rather than configurations) by indexing the refinement relation with a multiset.

Definition 5.2.3

1. $\leq^{\phi} = \bigcup\{\mathcal{R} \mid \mathcal{R}$ *is a strong* ϕ*-simulation* $\}$
2. $=^{\phi} = \leq^{\phi} \cap \leq^{\phi\,-1}$
3. $s \leq^{\phi}_{M} t$ *iff* $\langle s, M\rangle \leq^{\phi} \langle t, M\rangle$
4. $s =^{\phi}_{M} t$ *iff* $s \leq^{\phi}_{M} t$ *and* $t \leq^{\phi}_{M} s$

Lemma 5.2.4

1. $\leq^{\phi}$ *is the largest strong* ϕ*-simulation*
2. $\leq^{\phi}$ *is a partial order*
3. $=^{\phi}$ *is an equivalence relation*

Proof

1. By Lemma 5.2.2.3 $\leq^{\phi}$ is a strong ϕ-simulation. By Definition 5.2.3.1 it includes any other strong ϕ-simulation.
2. Reflexivity follows from Lemma 5.2.2.1, transitivity from Lemma 5.2.2.2, antisymmetry from Lemma 5.2.4.3.
3. Reflexivity and transitivity follow from Lemma 5.2.2.(1 and 2). Symmetry follows from Definition 5.2.3.2.

□

Analogously to [90] we use some fixed-point theory (see e.g [43]) to show that $\leq^{\phi}$ defines the relation that contains precisely all strong ϕ-simulations.

Definition 5.2.5 *Define a function* $\mathbb{F} : \mathbb{C} \times \mathbb{C} \to \mathbb{C} \times \mathbb{C}$ *as follows:*
If $\mathcal{R} \subseteq \mathbb{C}\times\mathbb{C}$*, then* $(\langle s, M\rangle, \langle t, M\rangle) \in \mathbb{F}(\mathcal{R})$ *if and only if, for all* λ*, for all* $M' : \phi(M, M')$*,*

1. $M = N$
2. $\langle s, M'\rangle \xrightarrow{\lambda} \langle s', M''\rangle \Rightarrow \langle t, M'\rangle \xrightarrow{\lambda} \langle t', M''\rangle$ *and* $(\langle s', M''\rangle, \langle t', M''\rangle) \in \mathcal{R}$
3. $s \equiv \mathsf{skip} \Rightarrow t \equiv \mathsf{skip}$

Lemma 5.2.6

1. *$\mathbb{F}$ is monotonic; i.e. if $\mathcal{R}_1 \subseteq \mathcal{R}_2$, then $\mathbb{F}(\mathcal{R}_1) \subseteq \mathbb{F}(\mathcal{R}_2)$.*

2. *$\mathcal{R}$ is a strong ϕ-simulation if and only if $\mathcal{R} \subseteq \mathbb{F}(\mathcal{R})$.*

Proof

1. Follows directly from Definition 5.2.5.

2. Follows directly from Definition 5.2.5 and Definition 5.2.1.

□

Monotonicity says that $\mathbb{F}$ preserves the ordering $\subseteq$ on $\mathbb{C} \times \mathbb{C}$. Strong ϕ-simulations are, by Lemma 5.2.6.2, exactly the pre-fixed-points of $\mathbb{F}$. We wish to show that $\leq^{\phi}$, which is the largest pre-fixed-point, is a fixed-point of $\mathbb{F}$.

Theorem 5.2.7 *$\leq^{\phi}$ is the largest fixed point of $\mathbb{F}$.*

Proof

- $\leq^{\phi} \subseteq \mathbb{F}(\leq^{\phi})$: By Lemma 5.2.4, $\leq^{\phi}$ is a strong ϕ-simulation. Then, by Lemma 5.2.6.2, follows $\leq^{\phi} \subseteq \mathbb{F}(\leq^{\phi})$.

- $\mathbb{F}(\leq^{\phi}) \subseteq \leq^{\phi}$: Monotonicity of $\mathbb{F}$ implies $\mathbb{F}(\leq^{\phi}) \subseteq \mathbb{F}(\mathbb{F}(\leq^{\phi}))$; i.e. $\mathbb{F}(\leq^{\phi})$ is a pre-fixed point of $\mathbb{F}$. But because $\leq^{\phi}$ is the largest pre-fixed point, it includes $\mathbb{F}(\leq^{\phi})$, i.e. $\mathbb{F}(\leq^{\phi}) \subseteq \leq^{\phi}$.

Moreover, $\leq^{\phi}$ must be the largest fixed point of $\mathbb{F}$, because it is the largest pre-fixed point. □

Hence $\leq^{\phi}$ is the largest relation that satisfies the definition of strong ϕ-simulation.

Next, we show that up-to simulations (as in [90]) can be defined for strong ϕ-simulation.

Definition 5.2.8 *Let $\mathcal{R} \subseteq \mathbb{C} \times \mathbb{C}$ and $\phi \subseteq \mathbb{M} \times \mathbb{M}$.*
We say that $\mathcal{R}$ is a strong ϕ-simulation up-to $\leq^{\phi}$ iff for all $(\langle s, M\rangle, \langle t, N\rangle) \in \mathcal{R}$, forall λ, forall $M' : \phi(M, M')$,

1. *$M = N$*

2. $\langle s, M'\rangle \xrightarrow{\lambda} \langle s', M''\rangle \Rightarrow \langle t, M'\rangle \xrightarrow{\lambda} \langle t', M''\rangle$ *and* $\langle s', M''\rangle \leq^{\phi} \mathcal{R} \leq^{\phi} \langle t', M''\rangle$

3. $s \equiv \mathsf{skip} \Rightarrow t \equiv \mathsf{skip}$

Lemma 5.2.9
If $\mathcal{R}$ is a strong ϕ-simulation up-to $\leq^{\phi}$, then $\leq^{\phi} \mathcal{R} \leq^{\phi}$ is a strong ϕ-simulation.

Proof Let $\langle s, M\rangle \leq^{\phi} \mathcal{R} \leq^{\phi} \langle t, N\rangle$, hence, for some s_1, t_1 and M_1, N_1, $\langle s, M\rangle \leq^{\phi} \langle s_1, M_1\rangle$, $\langle s_1, M_1\rangle \mathcal{R} \langle t_1, N_1\rangle$ and $\langle t_1, N_1\rangle \leq^{\phi} \langle t, N\rangle$. Because $\leq^{\phi}$ is a strong ϕ-simulation and $\mathcal{R}$ is a strong ϕ-simulation up-to $\leq^{\phi}$ follows $M = M_1 = N_1 = N$.
transition
Assume $\phi(M, M')$ and $\langle s, M'\rangle \xrightarrow{\lambda} \langle s', M''\rangle$.
From $\langle s, M\rangle \leq^{\phi} \langle s_1, M\rangle$ follows $\langle s_1, M'\rangle \xrightarrow{\lambda} \langle s_1', M''\rangle$ such that $\langle s', M''\rangle \leq^{\phi} \langle s_1', M''\rangle$.
From $\langle s_1, M\rangle \mathcal{R} \langle t_1, M\rangle$ follows $\langle t_1, M'\rangle \xrightarrow{\lambda} \langle t_1', M''\rangle$ such that $\langle s_1', M''\rangle \leq^{\phi} \mathcal{R} \leq^{\phi} \langle t_1', M''\rangle$.
From $\langle t_1, M\rangle \leq^{\phi} \langle t, M\rangle$ follows $\langle t, M'\rangle \xrightarrow{\lambda} \langle t', M''\rangle$ such that $\langle t_1', M''\rangle \leq^{\phi} \langle t', M''\rangle$.
Hence, $\langle s', M''\rangle \leq^{\phi} \leq^{\phi} \mathcal{R} \leq^{\phi} \leq^{\phi} \langle t', M''\rangle$.
By transitivity of $\leq^{\phi}$ follows $\langle s', M''\rangle \leq^{\phi} \mathcal{R} \leq^{\phi} \langle t', M''\rangle$.

termination: The proof is analogous to the above case. □

Lemma 5.2.10 *If $\mathcal{R}$ is a strong ϕ-simulation up-to $\leq^{\phi}$, then $\mathcal{R} \subseteq \leq^{\phi}$.*

Proof From Lemma 5.2.9 follows $\leq^{\phi} \mathcal{R} \leq^{\phi} \subseteq \leq^{\phi}$.
By reflexivity of $\leq^{\phi}$ (from Lemma 5.2.4.1) follows $Id_{\mathbb{C}} \subseteq \leq^{\phi}$, hence $\mathcal{R} \subseteq \leq^{\phi}$. □

We show how the statebased and stateless notions from Chapter 4 fit into the generic framework. This enables us to use the generic theory of refinement to fulfill some proof obligations regarding properties of statebased and stateless refinement. First, consider the statebased variant.

Theorem 5.2.11 *Let $\phi_{statebased} = Id_{\mathbb{M}}$. Then $\leqq = \leq^{\phi_{statebased}}$.*

Proof From $\phi_{statebased} = Id_{\mathbb{M}}$ follows $\{M' \mid (M, M') \in \phi_{statebased}\} = \{M\}$. Hence interference may only change a multiset M into M. This effectively means that interference is not allowed between successive transitions. The quantification $\forall M' : \phi_{statebased}(M, M')$ in Definition 5.2.1 reduces to $\forall M' : M = M'$ which then coincides precisely with the

definition statebased simulation. □

The basic properties of statebased simulation and statebased refinement promised by Proposition 4.3.2 and Proposition 4.3.4 follow immediately from Lemma 5.2.2 and Lemma 5.2.4.

From Theorem 5.2.7 follows that $\leqq$ is the largest relation that satisfies the definition of statebased simulation. Hence, $\leqq$ defines the relation that contains precisely all strong statebased simulations.

The fact that strong statebased simulation up-to $\leqq$ may be used to show strong statebased refinements, as promised by Proposition 4.3.6, follows from Lemma 5.2.10.

Next, we show that the stateless variant can be obtained as a special instance of ϕ-refinement.

Theorem 5.2.12 *Let* $\phi_{stateless} = \mathbb{M} \times \mathbb{M}$. *Then* $\{(\langle s, M\rangle, \langle t, M\rangle) \mid M \in \mathbb{M}\} = \leq^{\phi_{stateless}}$.

Proof From $\phi_{stateless} = \mathbb{M} \times \mathbb{M}$ follows $\{M' \mid (M, M') \in \phi\} = \mathbb{M}$. Hence the set of possible multisets that may result after interferences in a multiset M equals $\mathbb{M}$. Hence, the quantification $\forall M' : \phi(M, M')$ in Definition 5.2.1 can be written as $\forall M : M \in \mathbb{M}$ which then corresponds to the definition of stateless simulation – albeit that in the latter case the multiset component has been omitted from the (elements of the) simulation relation.

The correspondence between strong stateless simulation and strong $\mathbb{M} \times \mathbb{M}$-simulation is shown more formally by the following constructions. They show that every strong stateless refinement corresponds to a strong $\mathbb{M} \times \mathbb{M}$ refinement and vice versa.

Let $\mathcal{R}_1$ be a strong stateless simulation and let $\mathcal{R}_2$ be a strong $\mathbb{M} \times \mathbb{M}$-simulation. Define $\mathcal{R}'_1 = \{(\langle s, M\rangle, \langle s', M\rangle) \mid (s, s') \in \mathcal{R}_1\}$ and $\mathcal{R}'_2 = \{(s, s') \mid (\langle s, M\rangle, \langle s', M\rangle) \in \mathcal{R}_2\}$. It is straightforward to show that $\mathcal{R}'_1$ is a strong $\mathbb{M} \times \mathbb{M}$-simulation and $\mathcal{R}'_2$ is a strong stateless simulation. □

The basic properties attributed to stateless simulation and stateless refinement by Proposition 4.4.2 and Proposition 4.4.4 follow immediately from Lemma 5.2.2 and Lemma 5.2.4.

From Theorem 5.2.7 follows that $\leqslant$ is the largest relation that satisfies the definition of stateless simulation. Hence, $\leqslant$ defines the relation that contains precisely all strong stateless simulations.

The fact that strong stateless simulation up-to $\leqslant$ may be used to show strong stateless refinements, as promised by Proposition 4.4.7, follows from Lemma 5.2.10.

Theorem 5.2.13 shows that strong ϕ-refinement relations are ordered inversely by subset inclusion of the interference set. This can be interpreted as follows: if one configuration is a refinement of another configuration in some environment, then this refinement also holds in an environment which performs fewer interferences.

Theorem 5.2.13
Let $\phi, \psi \subseteq \mathbb{M} \times \mathbb{M}$ be binary relations over multisets. If $\phi \subseteq \psi$, then $\leq^{\psi} \subseteq \leq^{\phi}$.

Proof
Let $\mathcal{R} = \{(\langle s, M\rangle, \langle t, M\rangle) \mid \langle s, M\rangle \leq^{\psi} \langle t, M\rangle\}$.
We show that $\mathcal{R}$ is a strong ϕ-simulation. Assume $\langle s, M\rangle \mathcal{R} \langle t, M\rangle$ and $\phi(M, M')$.
transition
By $\phi \subseteq \psi$ follows $\psi(M, M')$. Hence if $\langle s, M'\rangle \xrightarrow{\lambda} \langle s', M''\rangle$, then by $\langle s, M\rangle \leq^{\psi} \langle t, M\rangle$ follows $\langle t, M'\rangle \xrightarrow{\lambda} \langle t', M''\rangle$ such that $\langle s', M''\rangle \leq^{\psi} \langle t', M''\rangle$. Hence $(\langle s', M''\rangle, \langle t', M''\rangle) \in \mathcal{R}$.
termination
If $s \equiv \mathsf{skip}$, then from $\langle s, M\rangle \leq^{\psi} \langle t, M\rangle$ follows $t \equiv \mathsf{skip}$. □

Theorem 5.2.13 has the following useful implication. Suppose we have two notions of refinement $\leq^{\phi}$ and $\leq^{\psi}$ such that $\phi \subseteq \psi$. If we have proven that $\langle s, M\rangle \leq^{\psi} \langle t, M\rangle$, then by Theorem 5.2.13 we may conclude $\langle s, M\rangle \leq^{\phi} \langle t, M\rangle$. Thus, to prove that some configurations are related by some notion of refinement, we may use any other notion of refinement that makes weaker assumptions about the environment. In particular this may be applied to statebased and stateless refinement.

Corollary 5.2.14 *If $s \leqslant t$, then $\langle s, M\rangle \leqq \langle t, M\rangle$ for all $M \in \mathbb{M}$.*

Proof By Theorem 5.2.13 from $Id_{\mathbb{M}} \subset \mathbb{M} \times \mathbb{M}$. □

5.3 Precongruence of Strong Generic Refinement

We are interested in deriving results about precongruence of strong ϕ-refinement. We know that, since statebased and stateless refinement are special cases of ϕ-refinement, some choices for ϕ yield precongruences and other choices do not. In this section, we will identify properties that ϕ must satisfy to ensure that $\leq^{\phi}$ is a precongruence.

To start with, we specify the domain over which we are considering precongruence of ϕ-refinement (formally: the carrier of the algebra). In principle, we can take any set $\mathbb{S}' \subseteq \mathbb{S}$ of schedules that satisfies the conditions that it is "closed" under the transition relation defined by the operational semantics for schedules.

Definition 5.3.1 *A set $\mathbb{S}'$ of schedules is* transition-closed *iff*

(S1) If $s \in \mathbb{S}'$ and $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$ for some M, M', then $s' \in \mathbb{S}'$.

Lemma 5.3.3 shows that any set of schedules that is limited to a fixed sort (i.e. a fixed set of rewrite rules) is transition-closed.

Definition 5.3.2 *Define, for sort L,*

$$\mathbb{S}_L = \{s \mid s \in \mathbb{S} \wedge \mathcal{L}(s) \subseteq L\}$$

Lemma 5.3.3 *For all sorts L, $\mathbb{S}_L$ is transition-closed.*

Proof By Lemma 3.3.17 and transitivity of $\subseteq$. □

In this section, we assume that $\mathbb{S}'$ is an arbitrary transition-closed set of schedules. Next, we introduce two criteria for the interference parameter ϕ.

Definition 5.3.4

(P1) For all $s \in \mathbb{S}'$, for all $M \in \mathbb{M}$, if $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$, then $\phi(M, M')$.

(P2) For all M, M', M'', if $\phi(M, M')$ and $\phi(M', M'')$, then $\phi(M, M'')$ (transitivity).

We give some intuition behind these criteria.

Suppose we want to refine a schedule t whose context consists of some schedule s; i.e. we consider $s \parallel t$. The schedule we want to put in place of t should behave as t under all possible interferences from s. This can be enforced by including all possible changes that s may make to the multiset in the interference set. Often we do not know precisely in which context a schedule is operating. However, if $\mathbb{S}'$ denotes the set of all possible schedules under consideration, then the interference must be due to a rewrite by some schedule s from $\mathbb{S}'$. Therefore, we consider any transition by any schedules from $\mathbb{S}'$ to be a potential interference. This is formalized by $(P1)$.

The transitivity condition $(P2)$ reflects the fact that the granularity or speed of interferences can not be observed. Suppose the environment may change the multiset

from M into M' and from M' into M''. If the intermediate multiset M' is not observed, then this pair of interferences has the same effect as a single interference that changes the multiset from M to M''.

A desirable property of a refinement in a system where interference may occur is that it remains valid if some interference changes the multiset. We introduce the notion of "*interference closedness*" which formalizes this notion of robustness. Suppose $\mathcal{R}$ is a simulation relation with $\langle s, M\rangle\mathcal{R}\langle t, M\rangle$. Relation $\mathcal{R}$ is interference closed if it guarantees that the behaviour of the left hand side can be mimicked by right hand side even if any interference from ϕ occurs.

Definition 5.3.5 *Let $\mathcal{R} \subseteq \mathbb{C}\times\mathbb{C}$. We say that $\mathcal{R}$ is* interference closed *if $\langle s, M\rangle\mathcal{R}\langle s', M\rangle$ and $\phi(M, M')$ implies $\langle s, M'\rangle\mathcal{R}\langle s', M'\rangle$.*

Lemma 5.3.6 shows that transitivity of ϕ implies that strong ϕ-refinement is interference closed.

Lemma 5.3.6 *If $\phi \subseteq \mathbb{M} \times \mathbb{M}$ is transitive, then $\leq^\phi$ is interference closed.*

Proof We have to show that if $\langle s, M\rangle \leq^\phi \langle t, M\rangle$ and $\phi(M, M')$, then $\langle s, M'\rangle \leq^\phi \langle t, M'\rangle$. Suppose $\phi(M', M'')$. Then by transitivity of ϕ follows $\phi(M, M'')$.
transition
Assume $\langle s, M''\rangle \xrightarrow{\lambda} \langle s', M'''\rangle$. Then by $\langle s, M\rangle \leq^\phi \langle t, M\rangle$ and $\phi(M, M'')$ follows $\langle t, M''\rangle \xrightarrow{\lambda} \langle t', M'''\rangle$ such that $\langle s', M'''\rangle \leq^\phi \langle t', M'''\rangle$.
termination
$s \equiv \mathsf{skip} \Rightarrow t \equiv \mathsf{skip}$ follows immediately from $\langle s, M\rangle \leq^\phi \langle t, M\rangle$. □

From transitivity of $\phi_{statebased} = Id_{\mathbb{M}}$ follows that all strong and weak statebased simulation relations are interference closed. Since $\phi_{stateless} = \mathbb{M} \times \mathbb{M}$ is transitive, all strong and weak stateless refinement relations are interference closed.

Next, we will show precongruence of ϕ-refinement by proving that ϕ-refinement is preserved by the combinators from our coordination language, provided ϕ satisfies conditions $(P1)$ and $(P2)$ from Definition 5.3.4. To this end, we first prove an auxiliary result which shows that if two schedules are structurally equivalent, then their behaviours are considered equivalent by any ϕ-refinement.

Lemma 5.3.7 *Let $\mathbb{S}'$ satisfy $(S1)$ and let ϕ satisfy $(P1)$ and $(P2)$. Let $s, t \in \mathbb{S}'$. If $s \equiv t$ then for all M, $s =^\phi_M t$.*

Proof By definition $s =^{\phi}_{M} t$ iff $s \leq^{\phi}_{M} t$ and $t \leq^{\phi}_{M} s$. Suppose $\phi(M, M')$.

- $s \equiv t \;\Rightarrow\; s \leq^{\phi}_{M} t$:
  transition
  If $\langle s, M' \rangle \xrightarrow{\lambda} \langle s', M'' \rangle$ then, by (N8) and $s \equiv t$ follows $\langle t, M' \rangle \xrightarrow{\lambda} \langle s', M'' \rangle$.
  By reflexivity of $\leq^{\phi}$ holds $s' \leq^{\phi}_{M''} s'$.
  termination
  If $s \equiv \mathsf{skip}$, then by transitivity of $\equiv$ follows $t \equiv \mathsf{skip}$.

- $s \equiv t \;\Rightarrow\; t \leq^{\phi}_{M} s$: The proof is analogous to the previous case.

□

Next, we show that the combinators from our coordination language preserve strong ϕ-refinement.

Lemma 5.3.8 *Let $\mathbb{S}'$ satisfy (S1) and let ϕ satisfy (P1) and (P2).*
Let $r, s_1, s_2, t_1, t_2 \in \mathbb{S}'$. If $s_1 \leq^{\phi}_{M} s_2$ and $t_1 \leq^{\phi}_{M} t_2$, then $r \to s_1[t_1] \leq^{\phi}_{M} r \to s_2[t_2]$.

Proof
Assume $\phi(M, M')$ and consider the following cases:
transition

- Suppose $\langle r \to s_1[t_1], M' \rangle \xrightarrow{\varepsilon} \langle t_1, M' \rangle$. Then by (N0) $\langle r \to s_2[t_2], M' \rangle \xrightarrow{\varepsilon} \langle t_2, M' \rangle$. From $t_1 \leq^{\phi}_{M} t_2$ and $\phi(M, M')$ follows, by Lemma 5.3.6, $t_1 \leq^{\phi}_{M'} t_2$.

- Suppose $\langle r \to s_1[t_1], M' \rangle \xrightarrow{\sigma} \langle s_1, M'' \rangle$. By (N1) $\langle r \to s_2[t_2], M' \rangle \xrightarrow{\sigma} \langle s_2, M'' \rangle$. By $(P1)$ follows $\phi(M', M'')$. Then, by $(P2)$ follows $\phi(M, M'')$. From $s_1 \leq^{\phi}_{M} s_2$ and Lemma 5.3.6 follows $s_1 \leq^{\phi}_{M''} s_2$.

termination
Holds vacuously. □

Lemma 5.3.9 *Let $\mathbb{S}'$ satisfy (S1) and let ϕ satisfy (P1) and (P2).*
Let $s_1, s_2, t_1, t_2 \in \mathbb{S}'$. If $s_1 \leq^{\phi}_{M} s_2$ and $t_1 \leq^{\phi}_{M} t_2$, then $s_1; t_1 \leq^{\phi}_{M} s_2; t_2$.

Proof Let $\mathcal{R} = \{(\langle s_1; t_1, M \rangle, \langle s_2; t_2, M \rangle) \mid s_1 \leq^{\phi}_{M} s_2, t_1 \leq^{\phi}_{M} t_2\}$.
We show that $\mathcal{R}$ is a strong ϕ-simulation *up-to* $\leq^{\phi}$.
transition

Assume $\phi(M, M')$ and $\langle s_1; t_1, M'\rangle \xrightarrow{\lambda} \langle s_1'; t_1', M''\rangle$. By $(P1)$ and $(P2)$ follows $\phi(M, M'')$. Consider the possible derivations

- By (N5) from $\langle s_1, M'\rangle \xrightarrow{\lambda} \langle s_1', M''\rangle$. From $s_1 \leq_M^\phi s_2$ follows $\langle s_2, M'\rangle \xrightarrow{\lambda} \langle s_2', M''\rangle$ such that $s_1' \leq_{M''}^\phi s_2'$. Then, by (N5), follows $\langle s_2; t_2, M'\rangle \xrightarrow{\lambda} \langle s_2'; t_2, M''\rangle$.
  From $t_1 \leq_M^\phi t_2$ and $\phi(M, M'')$ follows, by Lemma 5.3.6, that $t_1 \leq_{M''}^\phi t_2$.
  By $Id_{\mathbb{C}} \subseteq \ \leq^\phi$ follows $(\langle s_1'; t_1, M''\rangle, \langle s_2'; t_2, M''\rangle) \in \ \leq^\phi \mathcal{R} \leq^\phi$.

- By (N8) from $s_1 \equiv \mathsf{skip}$ and $\langle t_1, M'\rangle \xrightarrow{\lambda} \langle t_1', M''\rangle$. From $s_1 \leq_M^\phi s_2$ follows $s_2 \equiv \mathsf{skip}$. From $t_1 \leq_M^\phi t_2$ follows $\langle t_2, M'\rangle \xrightarrow{\lambda} \langle t_2', M''\rangle$ such that $t_1' \leq_{M''}^\phi t_2'$.
  By (N8) we derive $\langle s_2; t_2, M'\rangle \xrightarrow{\lambda} \langle t_2', M''\rangle$. From (E1) and Definition 5.2.3.2 follows, by Lemma 5.3.7, that $t_1' \leq_{M''}^\phi \mathsf{skip}; t_1'$ and $\mathsf{skip}; t_2' \leq_{M''}^\phi t_2'$. Hence from $(\mathsf{skip}; t_1', M'', \mathsf{skip}; t_2', M'') \in \mathcal{R}$ follows $(\langle t_1', M''\rangle, \langle t_2', M''\rangle) \in \ \leq^\phi \mathcal{R} \leq^\phi$.

termination
$s_1; t_1 \equiv \mathsf{skip}$ only if $s_1 \equiv \mathsf{skip}$ and $t_1 \equiv \mathsf{skip}$. From $s_1 \leq_M^\phi s_2$ and $t_1 \leq_M^\phi t_2$ then follows $s_2 \equiv \mathsf{skip}$ and $t_2 \equiv \mathsf{skip}$, hence $s_2; t_2 \equiv \mathsf{skip}$. □

Lemma 5.3.10 *Let $\mathbb{S}'$ satisfy $(S1)$ and let ϕ satisfy $(P1)$ and $(P2)$.*
Let $s_1, s_2, t_1, t_2 \in \mathbb{S}'$. If $s_1 \leq_M^\phi s_2$ and $t_1 \leq_M^\phi t_2$, then $s_1 \parallel t_1 \leq_M^\phi s_2 \parallel t_2$.

Proof Let $\mathcal{R} = \{(\langle s_1 \parallel t_1, M\rangle, \langle s_2 \parallel t_2, M\rangle) \mid s_1 \leq_M^\phi s_2, t_1 \leq_M^\phi t_2\}$.
We show that $\mathcal{R}$ is a strong ϕ-simulation by transition induction.
transition
Assume $\phi(M, M')$ and $\langle s_1 \parallel t_1, M'\rangle \xrightarrow{\lambda} \langle s_1' \parallel t_1', M''\rangle$. By $(P1)$ follows $\phi(M', M'')$. Then, by $(P2)$, follows $\phi(M, M'')$. Consider the different ways in which the last inference can be made:

- By (N2) from $\langle s_1, M'\rangle \xrightarrow{\lambda} \langle s_1', M''\rangle$. From $s_1 \leq_M^\phi s_2$ follows $\langle s_2, M'\rangle \xrightarrow{\lambda} \langle s_2', M''\rangle$ such that $s_1' \leq_{M''}^\phi s_2'$. By (N2) we derive $\langle s_2 \parallel t_2, M'\rangle \xrightarrow{\lambda} \langle s_2' \parallel t_2, M''\rangle$.
  By Lemma 5.3.6 we get from $t_1 \leq_M^\phi t_2$ and $\phi(M, M'')$ that $t_1 \leq_{M''}^\phi t_2$, hence $(\langle s_1' \parallel t_1, M''\rangle, \langle s_2' \parallel t_2, M''\rangle) \in \mathcal{R}$.

- By (N2) from $\langle t_1, M'\rangle \xrightarrow{\lambda} \langle t_1', M''\rangle$. The proof is analogous to the previous case.

- By (N3) from $\langle s_1, M'\rangle \xrightarrow{\lambda} \langle s_1', M''\rangle$ and $\langle t_1, M'\rangle \xrightarrow{\varepsilon} \langle t_1', M'\rangle$.
  From $s_1 \leq_M^\phi s_2$ follows $\langle s_2, M'\rangle \xrightarrow{\lambda} \langle s_2', M''\rangle$ such that $s_1 \leq_{M''}^\phi s_2$.
  From $t_1 \leq_M^\phi t_2$ follows $\langle t_2, M'\rangle \xrightarrow{\varepsilon} \langle t_2', M'\rangle$ such that $t_1' \leq_{M'}^\phi t_2'$.

By (N3) follows $\langle s_2 \parallel t_2, M' \rangle \xrightarrow{\lambda} \langle s_2' \parallel t_2', M'' \rangle$. From $\phi(M', M'')$ follows, by Lemma 5.3.6, that $t_1' \leq^{\phi}_{M''} t_2'$. Thus $(\langle s_1' \parallel t_1', M'' \rangle, \langle s_2' \parallel t_2', M'' \rangle) \in \mathcal{R}$.

- By (N3) from $\langle s_1, M' \rangle \xrightarrow{\varepsilon} \langle s_1', M' \rangle$ and $\langle t_1, M' \rangle \xrightarrow{\lambda} \langle t_1', M'' \rangle$.
  The proof is analogous to the previous case.

- By (N4) from $\langle s_1, M' \rangle \xrightarrow{\sigma_1} \langle s_1', M_1 \rangle$ with $\sigma_1 = N_1'/N_1$ and $\langle t_1, M' \rangle \xrightarrow{\sigma_2} \langle t_1', M_2 \rangle$ with $\sigma_2 = N_2'/N_2$ and $M' \models \sigma_1 \bowtie \sigma_2$.
  From $s_1 \leq^{\phi}_{M} s_2$ follows $\langle s_2, M' \rangle \xrightarrow{\sigma_1} \langle s_2', M_1 \rangle$ such that $s_1' \leq^{\phi}_{M_1} s_2'$.
  From $t_1 \leq^{\phi}_{M} t_2$ follows $\langle t_2, M' \rangle \xrightarrow{\sigma_2} \langle t_2', M_2 \rangle$ such that $t_1' \leq^{\phi}_{M_2} t_2'$.
  Then, by (N4), we derive $\langle s_2 \parallel t_2, M' \rangle \xrightarrow{\sigma} \langle s_2' \parallel t_2', M'' \rangle$.

  We need to show that $s_1' \leq^{\phi}_{M''} s_2'$ and $t_1' \leq^{\phi}_{M''} t_2'$.
  By (C0) follows that $N_1 \subseteq M'$ and $N_2 \subseteq M'$. Then, from $M' \models \sigma_1 \bowtie \sigma_2$ follows by Lemma A.2.6, that $\langle t_2, M_1 \rangle \xrightarrow{\sigma_2} \langle t_2', M'' \rangle$ and $\langle s_2, M_2 \rangle \xrightarrow{\sigma_1} \langle s_2', M'' \rangle$.
  Then, by $(P1)$, follows $\phi(M_1, M'')$ and $\phi(M_2, M'')$. By Lemma 5.3.6 follows $s_1' \leq^{\phi}_{M''} s_2'$ and $t_1' \leq^{\phi}_{M''} t_2'$. Hence $(\langle s_1' \parallel t_1', M'' \rangle, \langle s_2' \parallel t_2', M'' \rangle) \in \mathcal{R}$.

termination
$s_1 \parallel t_1 \equiv \mathsf{skip}$ only if $s_1 \equiv \mathsf{skip}$ and $t_1 \equiv \mathsf{skip}$. From $s_1 \leq^{\phi}_{M} s_2$ and $t_1 \leq^{\phi}_{M} t_2$ then follows $s_2 \equiv \mathsf{skip}$ and $t_2 \equiv \mathsf{skip}$, hence $s_2 \parallel t_2 \equiv \mathsf{skip}$. □

Lemma 5.3.11 *Let $\mathbb{S}'$ satisfy $(S1)$ and let ϕ satisfy $(P1)$ and $(P2)$.*
Let $s_1, s_2 \in \mathbb{S}'$. If $s_1 \leq^{\phi}_{M} s_2$ then $!s_1 \leq^{\phi}_{M} !s_2$.

Proof
Let $\mathcal{R} = \{(\langle t_1 \parallel !s_1, M \rangle, \langle t_2 \parallel !s_2, M \rangle) \mid t_1 \leq^{\phi}_{M} t_2,\ s_1 \leq^{\phi}_{M} s_2\} \cup Id_{\mathbb{S}}$.
We show that $\mathcal{R}$ is a strong ϕ-simulation up-to $\leq^{\phi}$ by induction on the depth of inference.
By Lemma 5.3.10 follows that $\mathcal{R}$ satisfies the following property.

$$\text{If } (\langle s_1, M \rangle, \langle s_2, M \rangle) \in \mathcal{R} \text{ and } t_1 \leq^{\phi}_{M} t_2, \text{ then } (\langle t_1 \parallel s_1, M \rangle, \langle t_2 \parallel s_2, M \rangle) \in \mathcal{R} \quad (*)$$

Suppose $\phi(M, M')$ and $\langle t_1 \parallel !s_1, M' \rangle \xrightarrow{\lambda} \langle s', M'' \rangle$. Then by $(P1)$ and $(P2)$ follows $\phi(M, M'')$. Consider the different ways in which the last step of the inference of the transition is done:

1. By (N2) from $\langle t_1, M' \rangle \xrightarrow{\lambda} \langle t_1', M'' \rangle$.
   From $t_1 \leq^{\phi}_{M} t_2$ follows $\langle t_2, M' \rangle \xrightarrow{\lambda} \langle t_2', M'' \rangle$ such that $t_1' \leq^{\phi}_{M''} t_2'$.

By (N2) we infer $\langle t_2 \parallel !s_2, M' \rangle \xrightarrow{\lambda} \langle t_2' \parallel !s_2, M'' \rangle$.
From $\phi(M, M'')$ and $s_1 \leq^{\phi}_{M} s_2$ we have by Lemma 5.3.6 that $s_1 \leq^{\phi}_{M''} s_2$.
Hence $(\langle t_1' \parallel !s_1, M'' \rangle, \langle t_2' \parallel !s_2, M'' \rangle) \in \leq^{\phi} \mathcal{R} \leq^{\phi}$.

2. By (N2) from $\langle !s_1, M' \rangle \xrightarrow{\lambda} \langle s_1', M'' \rangle$.
   This transition can be derived in the following ways.

   - By (N6) from $\langle s_1, M' \rangle \xrightarrow{\lambda} \langle s_1', M'' \rangle$.
     From $s_1 \leq^{\phi}_{M} s_2$ follows $\langle s_2, M' \rangle \xrightarrow{\lambda} \langle s_2', M'' \rangle$ such that $s_1' \leq^{\phi}_{M''} s_2'$.
     Then by (N6) $\langle !s_2, M' \rangle \xrightarrow{\lambda} \langle s_2', M'' \rangle$, and by (N2) $\langle t_2 \parallel !s_2, M' \rangle \xrightarrow{\lambda} \langle t_2 \parallel s_2', M'' \rangle$.
     From Lemma 5.3.6 follows $t_1 \leq^{\phi}_{M''} t_2$.
     By Lemma 5.3.10 we then get $t_1 \parallel s_1' \leq^{\phi}_{M''} t_2 \parallel s_2'$.
     From (E3) and (E8) follows by Lemma 5.3.7 that $t_1 \parallel s_1' \leq^{\phi} t_1 \parallel s_1' \parallel !\mathsf{skip}$ and $t_2 \parallel s_2' \parallel !\mathsf{skip} \leq^{\phi} t_2 \parallel s_2'$. Hence $(\langle t_1 \parallel s_1', M'' \rangle, \langle t_2 \parallel s_2', M'' \rangle) \in \leq^{\phi} \mathcal{R} \leq^{\phi}$.
   - By (N7) from $\langle s_1 \parallel !s_1, M' \rangle \xrightarrow{\lambda} \langle s_1', M'' \rangle$.
     By the induction hypothesis we get $\langle s_2 \parallel !s_2, M' \rangle \xrightarrow{\lambda} \langle s_2', M'' \rangle$ such that $(\langle s_1', M'' \rangle, \langle s_2', M'' \rangle) \in \leq^{\phi} \mathcal{R} \leq^{\phi}$. By (N7) we infer $\langle !s_2, M' \rangle \xrightarrow{\lambda} \langle s_2', M'' \rangle$.
     From (N2) we get $\langle t_2 \parallel !s_2, M' \rangle \xrightarrow{\lambda} \langle t_2 \parallel s_2', M'' \rangle$. Then, by Lemma 5.3.6 follows $t_1 \leq^{\phi}_{M''} t_2$. Hence from $(\langle s_1', M'' \rangle, \langle s_2', M'' \rangle) \in \mathcal{R}$ we get by (*) that $(\langle t_1 \parallel s_1', M'' \rangle, \langle t_2 \parallel s_2', M'' \rangle) \in \leq^{\phi} \mathcal{R} \leq^{\phi}$.

3. The proofs of the remaining cases

   - by (N3) from $\langle t_1, M' \rangle \xrightarrow{\lambda} \langle t_1', M'' \rangle$ and $\langle !s_1, M' \rangle \xrightarrow{\varepsilon} \langle s_1', M' \rangle$,
   - by (N3) from $\langle t_1, M' \rangle \xrightarrow{\varepsilon} \langle t_1', M' \rangle$ and $\langle !s_1, M' \rangle \xrightarrow{\lambda} \langle s_1', M'' \rangle$,
   - by (N4) from $\langle t_1, M' \rangle \xrightarrow{\sigma_1} \langle t_1', M_1 \rangle$ and $\langle !s_1, M' \rangle \xrightarrow{\sigma_2} \langle s', M_2 \rangle$ where $M' \models \sigma_1 \bowtie \sigma_2$.

   are routine combinations of cases 1. and 2. (analogous to the proof for parallel composition).

termination
$t_1 \parallel !s_1 \equiv \mathsf{skip}$ only if $t_1 \equiv \mathsf{skip}$ and $s_1 \equiv \mathsf{skip}$. From $t_1 \leq^{\phi} t_2$ and $s_1 \leq^{\phi} s_2$ follows $t_2 \equiv \mathsf{skip}$ and $s_2 \equiv \mathsf{skip}$. Hence $t_2 \parallel !s_2 \equiv \mathsf{skip}$. □

Lemma 5.3.12 *Let $\mathbb{S}'$ satisfy $(S1)$ and let ϕ satisfy $(P1)$ and $(P2)$.*
Let $s_1, s_2, t_1, t_2 \in \mathbb{S}'$. If $s_1 \leq^{\phi}_{M} s_2$ and $t_1 \leq^{\phi}_{M} t_2$, then $c \triangleright s_1[t_1] \leq^{\phi}_{M} c \triangleright s_2[t_2]$.

Proof The result follows from structural congruence and Lemma 5.3.6 by considering the cases $c = \mathit{true}$ and $c = \mathit{false}$. □

So far we have only dealt with refinement of ground schedules. We would also like to manipulate schedule expressions containing variables. Therefore, we extend the definition of ϕ-refinement to cover schedule expressions as follows.

Definition 5.3.13 *Let s_1 and $s_2 \in \mathbb{S}$ contain control variables $\overline{x}$ at most, and schedule variables $\overline{X}$ at most. Then $s_1 \leq^{\phi}_{M} s_2$ if, for all values $\overline{v}$ and ground schedules $\overline{t} \in \mathbb{S}_{ground}$, $\langle s_1[\overline{x} := \overline{v}]\{\overline{t}/\overline{X}\}, M\rangle \leq^{\phi} \langle s_2[\overline{x} := \overline{v}]\{\overline{t}/\overline{X}\}, M\rangle$.*

The equivalence $=^{\phi}_{M}$ is extended analogous to Definition 5.3.13. We proceed by showing that recursive definitions preserve equivalence.

Lemma 5.3.14 *If $S(\overline{x}) \mathrel{\widehat{=}} s$, then for all ϕ, M, $S(\overline{x}) =^{\phi}_{M} s$.*

Proof

- $S(\overline{x}) \leq^{\phi}_{M} s$: Suppose $\phi(M, M')$.
  transition
  For any $\overline{v}$, a transition $\langle S(\overline{v}), M'\rangle \xrightarrow{\lambda} \langle s', M''\rangle$ is derived by (E9) and (N8) from $\langle s[\overline{x} := \overline{v}], M'\rangle \xrightarrow{\lambda} \langle s', M''\rangle$. By reflexivity of $\leq^{\phi}$ follows $s' \leq^{\phi}_{M''} s'$.
  termination
  Follows from (E9) and transitivity of $\equiv$.

- $s \leq^{\phi}_{M} S(\overline{x})$: The proof is analogous to the previous case.

□

Lemma 5.3.15 proves that if s_1 is a generic refinement of s_2 (in the sense of Definition 5.3.13), then a schedule that invokes s_1 recursively is a refinement of a schedule that invokes s_2 recursively. This essentially proves the monotonicity of building recursive schedules with respect to the refinement relation. The control variables play no role of importance in Lemma 5.3.15 and have been left out to increase readability.

Lemma 5.3.15 *Let s_1 and s_2 contain at most schedule variable X. Let $S_1, S_2 \in \mathcal{S}$ be schedule identifiers defined by $S_1 \mathrel{\widehat{=}} s_1\{S_1/X\}$ and $S_2 \mathrel{\widehat{=}} s_2\{S_2/X\}$. If $s_1 \leq^{\phi}_{M} s_2$, then $S_1 \leq^{\phi}_{M} S_2$.*

Proof We show that

$$\mathcal{R} = \{(\langle t\{S_1/X\}, M\rangle, \langle t\{S_2/X\}, M\rangle) \mid t \text{ contains at most the variable } X\}$$

is a strong ϕ-simulation up-to $\leq^{\phi}$. Suppose $\phi(M, M')$. We first prove the termination case because this will be needed in the transition case. The termination case is proven by induction on the structure of t; the transition case by induction on the depth of the inference of an arbitrary transition $\langle t\{S_1/X\}, M'\rangle \xrightarrow{\lambda} \langle s', M''\rangle$.
termination

We must show that $t\{S_1/X\} \equiv \mathsf{skip} \Rightarrow t\{S_2/X\} \equiv \mathsf{skip}$.
To this end, we proceed by induction on the structure of t:

- $t \equiv \mathsf{skip}$:
  Then $t\{S_1/X\} \equiv \mathsf{skip} \equiv t\{S_2/X\}$.

- $t \equiv X$:
  Then $t\{S_1/X\} \equiv S_1$ and $t\{S_2/X\} \equiv S_2$. From $t\{S_1/X\} \equiv \mathsf{skip}$ follows, by (E9), that $s_1\{S_1/X\} \equiv \mathsf{skip}$. By $s_1 \leq^{\phi} s_2$ follows $s_2\{S_2/X\} \equiv \mathsf{skip}$. By (E9) we infer $S_2 \equiv \mathsf{skip}$, hence $t\{S_2/X\} \equiv \mathsf{skip}$.

- $t \equiv r \to t_1[t_2]$:
  Holds vacuously.

- $t \equiv c \triangleright t_1[t_2]$:
  Then $t\{S_1/X\} \equiv c \triangleright t_1\{S_1/X\}[t_2\{S_1/X\}]$ and $t\{S_2/X\} \equiv c \triangleright t_1\{S_2/X\}[t_2\{S_2/X\}]$.

  - If $c = \mathit{true}$ then $t_1\{S_1/X\} \equiv \mathsf{skip}$.
    By the induction hypothesis $t_1\{S_2/X\} \equiv \mathsf{skip}$, hence $t\{S_2/X\} \equiv \mathsf{skip}$.
  - If $c = \mathit{false}$ the proof proceeds analogously.

- $t \equiv t_1 \parallel t_2$:
  Then $t_1\{S_1/X\} \parallel t_2\{S_1/X\} \equiv \mathsf{skip}$ only if, by (E1), $t_1\{S_1/X\} \equiv \mathsf{skip}$ and $t_2\{S_1/X\} \equiv \mathsf{skip}$. By the induction hypothesis follow $t_1\{S_2/X\} \equiv \mathsf{skip}$ and $t_2\{S_2/X\} \equiv \mathsf{skip}$. By (E1) we conclude $t_1\{S_2/X\} \parallel t_2\{S_2/X\} \equiv \mathsf{skip}$.

- $t \equiv t_1; t_2$:
  Analogous to the previous case.

- $t \equiv !t'$:
  Then $!t'\{S_1/X\} \equiv \mathsf{skip}$ only if, by (E8), $t'\{S_1/X\} \equiv \mathsf{skip}$. By the induction hypothesis follows $t'\{S_2/X\} \equiv \mathsf{skip}$. By (E8) we conclude $!t\{S_2/X\} \equiv \mathsf{skip}$.

- $t \equiv T$, where $T \mathrel{\widehat{=}} t'$ and t' is a schedule without variables. Then $t'\{S_1/X\} \equiv \mathsf{skip}$ only if $t' \equiv \mathsf{skip}$. Because t' contains no variables, $t\{S_1/X\} \equiv t\{S_2/X\} \equiv t'$. By (E9) and transitivity of $\equiv$ follows $t\{S_2/X\} \equiv \mathsf{skip}$.

transition
Consider the possible transitions for $\langle t, M'\rangle \xrightarrow{\lambda} \langle t', M''\rangle$ where t is one of the following:

- $t \equiv X$:
  Then $t\{S_1/X\} \equiv S_1$, hence the transition we consider is $\langle S_1, M'\rangle \xrightarrow{\lambda} \langle s', M''\rangle$. This is derived, by (N8) and (E9), from $\langle s_1\{S_1/X\}, M'\rangle \xrightarrow{\lambda} \langle s', M''\rangle$. The derivation of the latter transition is shorter, hence from the induction hypothesis follows $\langle s_1\{S_2/X\}, M'\rangle \xrightarrow{\lambda} \langle s'', M''\rangle$ with $(\langle s', M''\rangle, \langle s'', M''\rangle) \in \leq^{\phi} \mathcal{R} \leq^{\phi}$.
  From $s_1 \leq^{\phi}_{M} s_2$ and $\phi(M, M')$ follows $\langle s_2\{S_2/X\}, M'\rangle \xrightarrow{\lambda} \langle s''', M''\rangle$ with $(\langle s'', M''\rangle, \langle s''', M''\rangle) \in \leq^{\phi}$. Because $S_2 \mathrel{\widehat{=}} s_2\{S_2/X\}$ and $S_2 \equiv t\{S_2/X\}$ we get, by (N8), $\langle t\{S_2/X\}, M'\rangle \xrightarrow{\lambda} \langle s''', M''\rangle$ with $(\langle s', M''\rangle, \langle s''', M''\rangle) \in \leq^{\phi} \mathcal{R} \leq^{\phi}$ as required.

- $t \equiv r \to t_1[t_2]$:
  Then $t\{S_1/X\} \equiv r \to t_1\{S_1/X\}[t_2\{S_1/X\}]$. Here t_1 and t_2 contain at most the variable X. The transitions we have to consider are

  - $\langle r \to t_1\{S_1/X\}[t_2\{S_1/X\}], M'\rangle \xrightarrow{\varepsilon} \langle t_2\{S_1/X\}, M''\rangle$: then by (N0) also $\langle r \to t_1\{S_2/X\}[t_2\{S_2/X\}], M'\rangle \xrightarrow{\varepsilon} \langle t_2\{S_2/X\}, M''\rangle$. By reflexivity of $\leq^{\phi}$ and by definition of $\mathcal{R}$, follows $(\langle t_2\{S_1/X\}, M''\rangle, \langle t_2\{S_2/X\}, M''\rangle) \in \leq^{\phi} \mathcal{R} \leq^{\phi}$.
  - $\langle r \to t_1\{S_1/X\}[t_2\{S_1/X\}], M'\rangle \xrightarrow{\sigma} \langle t_1\{S_1/X\}, M''\rangle$:
    The proof is analogous to the previous case.

- $t \equiv c \triangleright t_1[t_2]$:
  Then $t\{S_1/X\} \equiv c \triangleright t_1\{S_1/X\}[t_2\{S_1/X\}]$ and $t\{S_2/X\} \equiv c \triangleright t_1\{S_2/X\}[t_2\{S_2/X\}]$. Consider the cases $c = \mathit{true}$ and $c = \mathit{false}$.

  - $c = \mathit{true}$: A transition can be derived by (N8) from $c \triangleright t_1[t_2] \equiv t_1$ and $\langle t_1\{S_1/X\}, M'\rangle \xrightarrow{\lambda} \langle t_1', M''\rangle$. This transition is derived by a shorter inference,

hence by the induction hypothesis we get $\langle t_1\{S_2/X\}, M'\rangle \xrightarrow{\lambda} \langle t_1'', M''\rangle$ such that $(\langle t_1', M''\rangle, \langle t_1'', M''\rangle) \in \leq^{\phi} \mathcal{R} \leq^{\phi}$.
By (N8) we derive $\langle c \triangleright t_1[t_2]\{S_2/X\}, M'\rangle \xrightarrow{\lambda} \langle t_1'', M''\rangle$.

– $c = \mathit{false}$: Analogous to the case $c = \mathit{true}$.

- $t \equiv t_1; t_2$:
Then $t\{S_1/X\} \equiv t_1\{S_1/X\}; t_2\{S_1/X\}$.
There are two possibilities for deriving a transition:

  – By (N5) from $\langle t_1\{S_1/X\}, M'\rangle \xrightarrow{\lambda} \langle t_1', M''\rangle$, hence $t' \equiv t_1'; t_2\{S_1/X\}$.
  This is derived by a shorter inference, so by the induction hypothesis follows $\langle t_1\{S_2/X\}, M'\rangle \xrightarrow{\lambda} \langle t_1'', M''\rangle$ such that $(\langle t_1', M''\rangle, \langle t_1'', M''\rangle) \in \leq^{\phi} \mathcal{R} \leq^{\phi}$.
  Then by (N5) follows $\langle t_1\{S_2/X\}; t_2\{S_2/X\}, M'\rangle \xrightarrow{\lambda} \langle t_1''; t_2\{S_2/X\}, M''\rangle$.
  From $(\langle t_1', M''\rangle, \langle t_1'', M''\rangle) \in \leq^{\phi} \mathcal{R} \leq^{\phi}$ follows that there are g and g' such that $\langle t_1', M''\rangle \leq^{\phi} \langle g, M''\rangle$, $(\langle g, M''\rangle, \langle g', M''\rangle) \in \mathcal{R}$ and $\langle g', M''\rangle \leq^{\phi} \langle t_1'', M''\rangle$.
  Then by Lemma 5.3.9 follows that $t_1'; t_2\{S_2/X\} \leq^{\phi}_{M''} g; t_2\{S_2/X\}$ and $g'; t_2\{S_2/X\} \leq^{\phi}_{M''} t_1''; t_2\{S_2/X\}$.
  Because t_2 contains at most variable X, we get by definition of $\mathcal{R}$ that $(\langle g; t_2\{S_2/X\}, M''\rangle, \langle g'; t_2\{S_2/X\}, M''\rangle) \in \mathcal{R}$.
  Hence $(\langle t_1'; t_2\{S_2/X\}, M''\rangle, \langle t_1''; t_2\{S_2/X\}, M''\rangle) \in \leq^{\phi} \mathcal{R} \leq^{\phi}$ as required.

  – By (N8) from $t_1\{S_1/X\} \equiv \mathsf{skip}$ and $\langle t_2\{S_1/X\}, M'\rangle \xrightarrow{\lambda} \langle t_2', M''\rangle$, hence $t' \equiv t_2'$. By the *termination*-part of this proof we know that $t_1\{S_2/X\} \equiv \mathsf{skip}$.
  This transition is derived by a shorter inference, so by the induction hypothesis that $\langle t_2\{S_2/X\}, M'\rangle \xrightarrow{\lambda} \langle t_2'', M''\rangle$ such that $(\langle t_2', M''\rangle, \langle t_2'', M''\rangle) \in \leq^{\phi} \mathcal{R} \leq^{\phi}$.
  By (N8) we infer $\langle t_1\{S_2/X\}; t_2\{S_2/X\}, M'\rangle \xrightarrow{\lambda} \langle t_2'', M''\rangle$.

- $t \equiv t_1 \parallel t_2$:
Then $t\{S_1/X\} \equiv t_1\{S_1/X\} \parallel t_2\{S_1/X\}$ and a transition can be derived by

  – (N2) from $\langle t_1\{S_1/X\}, M'\rangle \xrightarrow{\lambda} \langle t_1', M''\rangle$. By the induction hypothesis follows $\langle t_1\{S_2/X\}, M'\rangle \xrightarrow{\lambda} \langle t_1'', M''\rangle$ such that $(\langle t_1', M''\rangle, \langle t_1'', M''\rangle) \in \leq^{\phi} \mathcal{R} \leq^{\phi}$.
  Then, we derive, by (N2), $\langle t_1\{S_2/X\} \parallel t_2\{S_2/X\}, M'\rangle \xrightarrow{\lambda} \langle t_1'' \parallel t_2\{S_2/X\}, M''\rangle$.
  From $(\langle t_1', M''\rangle, \langle t_1'', M''\rangle) \in \leq^{\phi} \mathcal{R} \leq^{\phi}$ follows that there are g and g' that contain at most variable X such that $\langle t_1', M''\rangle \leq^{\phi} \langle g, M''\rangle$, $(\langle g, M''\rangle, \langle g', M''\rangle) \in \mathcal{R}$ and $\langle g', M''\rangle \leq^{\phi} \langle t_1'', M''\rangle$. By Lemma 5.3.10 follows $t_1' \parallel t_2\{S_2/X\} \leq^{\phi}_{M''} g \parallel t_2\{S_2/X\}$ and $g' \parallel t_2\{S_2/X\} \leq^{\phi}_{M''} t_1'' \parallel t_2\{S_2/X\}$.

Because t_2 contains at most variable X, we get by definition of $\mathcal{R}$ that $(\langle g \parallel t_2\{S_2/X\}, M''\rangle, \langle g' \parallel t_2\{S_2/X\}, M''\rangle) \in \mathcal{R}$.
Hence $(\langle t_1' \parallel t_2\{S_2/X\}, M''\rangle, \langle t_1'' \parallel t_2\{S_2/X\}, M''\rangle) \in \leq^\phi \mathcal{R} \leq^\phi$ as required.

- (N2) from $\langle t_2\{S_1/X\}, M'\rangle \xrightarrow{\lambda} \langle t_2', M''\rangle$.
  The proof is analogous to the previous case.

- (N3) from $\langle t_1\{S_1/X\}, M'\rangle \xrightarrow{\lambda} \langle t_1', M''\rangle$ and $\langle t_2\{S_1/X\}, M'\rangle \xrightarrow{\varepsilon} \langle t_2', M'\rangle$. The induction hypothesis applies to both of these transitions. This yields $\langle t_1\{S_2/X\}, M'\rangle \xrightarrow{\lambda} \langle t_1'', M''\rangle$ such that $\langle t_1', M''\rangle \leq^\phi \mathcal{R} \leq^\phi \langle t_1'', M''\rangle$ for the former, and $\langle t_2\{S_2/X\}, M'\rangle \xrightarrow{\varepsilon} \langle t_2'', M'\rangle$ such that $\langle t_2', M'\rangle \leq^\phi \mathcal{R} \leq^\phi \langle t_2'', M'\rangle$ for the latter transition. Hence there are g, g', h, h' that contain at most variable X such that $t_1' \leq^\phi_{M''} g$, $(\langle g, M''\rangle, \langle g', M''\rangle) \in \mathcal{R}$, $g' \leq^\phi_{M''} t_1''$ and $t_2' \leq^\phi_{M'} h$, $(\langle h, M'\rangle, \langle h', M'\rangle) \in \mathcal{R}$ and $h' \leq^\phi_{M'} t_2''$. From the transition by t_1 follows by $(P1)$ that $\phi(M', M'')$. By Lemma 5.3.6 then follows $t_2' \leq^\phi_{M''} h$ and $h' \leq^\phi_{M''} t_2''$. By Lemma 5.3.10 follows $t_1' \parallel t_2' \leq^\phi_{M''} g \parallel h$, $g' \parallel h' \leq^\phi_{M''} t_1'' \parallel t_2''$. By definition of $\mathcal{R}$ we have $(\langle g \parallel h, M''\rangle, \langle g' \parallel h', M''\rangle) \in \mathcal{R}$, hence $(\langle t_1' \parallel t_2', M''\rangle, \langle t_1'' \parallel t_2'', M''\rangle) \in \leq^\phi \mathcal{R} \leq^\phi$.

- (N3) from $\langle t_1\{S_1/X\}, M'\rangle \xrightarrow{\varepsilon} \langle t_1', M'\rangle$ and $\langle t_2\{S_1/X\}, M'\rangle \xrightarrow{\lambda} \langle t_2', M''\rangle$.
  The proof is analogous to the previous case.

- (N4) from $\langle t_1\{S_1/X\}, M'\rangle \xrightarrow{\sigma_1} \langle t_1', M_1\rangle$ and $\langle t_2\{S_1/X\}, M'\rangle \xrightarrow{\sigma_2} \langle t_2', M_2\rangle$ where $M' \models \sigma_1 \bowtie \sigma_2$. From the induction hypothesis follows $\langle t_1\{S_2/X\}, M'\rangle \xrightarrow{\sigma_1} \langle t_1'', M_1\rangle$ and $\langle t_2\{S_2/X\}, M'\rangle \xrightarrow{\sigma_2} \langle t_2'', M_2\rangle$ such that $\langle t_1', M_1\rangle \leq^\phi \mathcal{R} \leq^\phi \langle t_1'', M_1\rangle$ and
  $\langle t_2', M_2\rangle \leq^\phi \mathcal{R} \leq^\phi \langle t_2'', M_2\rangle$.

  Hence there are g, g', h, h' that contain at most variable X such that $t_1' \leq^\phi_{M_1} g$, $(\langle g, M_1\rangle, \langle g', M_1\rangle) \in \mathcal{R}$, $g' \leq^\phi_{M_1} t_1''$ and $t_2' \leq^\phi_{M_2} h$, $(\langle h, M_2\rangle, \langle h', M_2\rangle) \in \mathcal{R}$ and $h' \leq^\phi_{M_2} t_2''$. By Lemma A.2.6 follows that execution of σ_1 and σ_2 may be interleaved in arbitrary order; hence $\langle t_1\{S_2/X\}, M_2\rangle \xrightarrow{\sigma_1} \langle t_1'', M''\rangle$ and $\langle t_2\{S_2/X\}, M_1\rangle \xrightarrow{\sigma_2} \langle t_2'', M''\rangle$. From these transitions follows by $(P1)$ that $\phi(M_1, M'')$ and $\phi(M_2, M'')$. By Lemma 5.3.6 follows $t_1' \leq^\phi_{M''} g$, $g' \leq^\phi_{M''} t_1''$ and $t_2' \leq^\phi_{M''} h$, $h' \leq^\phi_{M''} t_2''$. By Lemma 5.3.10 follows $t_1' \parallel t_2' \leq^\phi_{M''} g \parallel h$ and $g' \parallel h' \leq^\phi_{M''} t_1'' \parallel t_2''$. By definition of $\mathcal{R}$ we have $(\langle g \parallel h, M''\rangle, \langle g' \parallel h', M''\rangle) \in \mathcal{R}$, hence $(\langle t_1' \parallel t_2', M''\rangle, \langle t_1'' \parallel t_2'', M''\rangle) \in \leq^\phi \mathcal{R} \leq^\phi$.

- $t \equiv !t'$:

Then $t\{S_1/X\} \equiv !t'\{S_1/X\}$. A transition can be derived in the following ways:

- By (N6) from $\langle t'\{S_1/X\}, M'\rangle \xrightarrow{\lambda} \langle t'', M''\rangle$.
  The term t' contains at most the variable X, and the transition is derived by a shorter inference hence the induction hypothesis gives $\langle t'\{S_2/X\}, M'\rangle \xrightarrow{\lambda} \langle t''', M''\rangle$ such that $(\langle t'', M''\rangle, \langle t''', M''\rangle) \in \leq^{\phi} \mathcal{R} \leq^{\phi}$.
  By (N6) $\langle !t'\{S_2/X\}, M'\rangle \xrightarrow{\lambda} \langle t''', M''\rangle$.
- By (N7) from $\langle t'\{S_1/X\} \parallel !t'\{S_1/X\}, M'\rangle \xrightarrow{\lambda} \langle t'', M''\rangle$.
  Because $t \equiv !t'$ contains at most variable X, so does t', hence also $t' \parallel !t'$.
  The transition is derived by a shorter inference, hence the induction hypothesis gives $\langle t'\{S_2/X\} \parallel !t'\{S_2/X\}, M'\rangle \xrightarrow{\lambda} \langle t''', M''\rangle$ such that $(\langle t'', M''\rangle, \langle t''', M''\rangle) \in \leq^{\phi} \mathcal{R} \leq^{\phi}$. By (N7) $\langle !t'\{S_2/X\}, M'\rangle \xrightarrow{\lambda} \langle t''', M''\rangle$.

- $t \equiv T$, where $T \mathrel{\widehat{=}} t'$ and t' is a ground schedule. Then $t\{S_1/X\} \equiv t\{S_2/X\} \equiv t'$.
  If $\langle t\{S_1/X\}, M'\rangle \xrightarrow{\lambda} \langle t'', M''\rangle$ then $\langle t\{S_2/X\}, M'\rangle \xrightarrow{\lambda} \langle t'', M''\rangle$.
  From $(t'', t'') \equiv (t''\{S_1/X\}, t''\{S_2/X\})$ and reflexivity of $\leq^{\phi}$ follows $(\langle t'', M''\rangle, \langle t'', M''\rangle) \in \leq^{\phi} \mathcal{R} \leq^{\phi}$.

□

Theorem 5.3.16 *Let $\mathbb{S}'$ be a transition closed set of schedules (satisfy $(S1)$ from Definition 5.3.1) and let ϕ satisfy $(P1)$ and $(P2)$ (from Definition 5.3.4). Then $\leq^{\phi}_{M}$ is a precongruence on $\mathbb{S}'$.*

Proof From Lemmas 5.3.8, 5.3.9, 5.3.10, 5.3.11, 5.3.12 and 5.3.15. □

It follows that $=^{\phi}$ is a congruence on schedules.

Corollary 5.3.17 *$=^{\phi}_{M}$ is a congruence relation on schedules.*

Proof Straightforward using Definition 5.2.3.2. □

Theorem 5.3.16 implies the precongruence of strong stateless refinement.

Corollary 5.3.18 *$\leqslant$ is a precongruence on $\mathbb{S}$.*

Proof Clearly $\mathbb{S}$ satisfies (S1) and $\phi_{stateless}$ satisfies $(P1)$ and $(P2)$. □

5.4 Soundness of Strong Generic Refinement

If we use refinement to replace one schedule by another, we want it to preserve the set of outcomes. Generally, the outcome of a schedule depends on the interference. However, this is not taken into account by the definition of the capability function of Definition 3.2.3. We propose the following adaptation of the capability function which does take interference into account.

The following definitions assume that transitions of the schedules are atomic and that interference may take place between transitions. Hence, an observer could see the following sequence of modifications to the multiset.

$$\langle s, M\rangle \underbrace{\quad}_{\phi(M,M')} \langle s, M'\rangle \xrightarrow{\lambda_1} \langle s_1, M_1\rangle \underbrace{\quad}_{\phi(M_1,M_1')} \cdots \underbrace{\quad}_{\phi(M_n,M_n')} \langle s_n, M_n'\rangle \xrightarrow{\lambda_{n+1}} \langle \mathsf{skip}, M_{n+1}\rangle \underbrace{\quad}_{\phi(M_n,M_{n+1}')}$$

The alternation of actions of a schedule and of the interference leads to the following definitions of divergence and termination.

Definition 5.4.1 *A configuration $\langle s, M\rangle$ may diverge under interference ϕ, denoted $\langle s, M\rangle\uparrow^\phi$, if and only if $\langle s, M\rangle = \langle s_0, M_0\rangle$ and for all $i \geq 0$ there exists a λ_i, M_i' and M_{i+1} such that $\phi(M_i, M_i')$ and $\langle s_i, M_i'\rangle \xrightarrow{\lambda_i} \langle s_{i+1}, M_{i+1}\rangle$.*

Definition 5.4.2 *A configuration $\langle s, M\rangle$ may terminate in M' under interference ϕ, denoted $\langle s, M\rangle\downarrow^\phi M'$, if and only if there exists some $n \in \mathbb{N}$ such that there exists $\lambda_0, \ldots, \lambda_{n-1}$, $M_0, \ldots, M_n$ and $M_0', \ldots, M_n'$ such that $\langle s, M\rangle = \langle s_0, M_0\rangle$ and for all $i : 0 \leq i < n : \phi(M_i, M_i') \wedge \langle s_i, M_i'\rangle \xrightarrow{\lambda_i} \langle s_{i+1}, M_{i+1}\rangle$ and $\phi(M_{i+1}, M_{i+1}')$ where $\langle s_n, M_n'\rangle = \langle \mathsf{skip}, M'\rangle$.*

Definition 5.4.3 *The capability function $\mathcal{C} : \mathbb{S} \times \mathbb{M} \to \mathcal{P}(\mathbb{M}) \cup \{\bot\}$ for schedules, is defined as*

$$\mathcal{C}^\phi(s, M) = \{\bot \mid \langle s, M\rangle\uparrow^\phi\} \cup \{M' \mid \langle s, M\rangle\downarrow^\phi M'\}$$

Theorem 5.4.4 shows that generic refinement is sound in the sense that it ensures that any output that a refining configuration may yield is an output that we were willing to accept from the original configuration.

Theorem 5.4.4 *If $s \leq^\phi_M t$, then $\mathcal{C}^\phi(s, M) \subseteq \mathcal{C}^\phi(t, M)$.*

Proof Consider the following cases

- $\bot \in \mathcal{C}^\phi(s, M)$: hence $\langle s, M\rangle\uparrow^\phi$. From $s \leq^\phi_M t$ follows $\langle t, M\rangle\uparrow^\phi$, hence $\bot \in \mathcal{C}^\phi(t, M)$.
- $M' \in \mathcal{C}^\phi(s, M)$: hence $\langle s, M\rangle\downarrow^\phi M'$. From $s \leq^\phi_M t$ follows $\langle t, M\rangle\downarrow^\phi M'$, hence $M' \in \mathcal{C}^\phi(t, M)$.

□

In a number of cases we are only interested in the output of a configuration if it terminates. This is captured by the generic output function (which ignores the possibility of divergence).

Definition 5.4.5 *Let $\phi \subseteq \mathbb{M} \times \mathbb{M}$ be an interference set. The* output *of a configuration $\langle t, M\rangle$ under interference ϕ, denoted $\mathcal{O}^\phi(t, M)$, is defined by*

$$\mathcal{O}^\phi(s, M) = \{M' \mid \langle s, M\rangle\downarrow^\phi M'\}$$

We show that the set of possible outcomes $\mathcal{O}^\phi(t, M)$ never increases, but possibly decreases, as execution progresses (progress of execution is taken to be either a transition by a schedule, or an interference from the context)

Lemma 5.4.6 *Let $\phi \subseteq \mathbb{M} \times \mathbb{M}$ be a reflexive and transitive interference set. Let $\langle t, M\rangle$ be a configuration.*

1. *If $\phi(M, M')$, then $\mathcal{O}^\phi(t, M') \subseteq \mathcal{O}^\phi(t, M)$.*
2. *If $\langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle$, then $\mathcal{O}^\phi(t', M') \subseteq \mathcal{O}^\phi(t, M)$.*

Proof We will use the following property of $\downarrow^\phi$

$$\langle t, M\rangle\downarrow^\phi M' \Leftrightarrow (\exists N, N' : \phi(M, N) \wedge \langle t, N\rangle \xrightarrow{\lambda} \langle t', N'\rangle \wedge \langle t', N'\rangle\downarrow^\phi M') \quad (*)$$

1. Suppose $\phi(M, M')$ and $M'' \in \mathcal{O}^\phi(t, M')$. Then $\langle t, M'\rangle\downarrow^\phi M''$, hence by (*) follows

   $$(\exists N, N' : \phi(M', N) \wedge \langle t, N\rangle \xrightarrow{\lambda} \langle t', N'\rangle \wedge \langle t', N'\rangle\downarrow^\phi M'')$$

   From transitivity of ϕ follows from $\phi(M, M')$ and $\phi(M', N)$ that $\phi(M, N)$. Hence

   $$(\exists N, N' : \phi(M, N) \wedge \langle t, N\rangle \xrightarrow{\lambda} \langle t', N'\rangle \wedge \langle t', N'\rangle\downarrow^\phi M'')$$

   Then, by (*) follows $\langle t, M\rangle\downarrow^\phi M''$, hence $M'' \in \mathcal{O}^\phi(t, M)$.

2. Suppose $\langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle$ and $M'' \in \mathcal{O}^{\phi}(t', M')$. From the latter follows $\langle t', M'\rangle{\downarrow^{\phi}}M''$. By reflexivity of ϕ follows $\phi(M, M)$. Hence, we have

$$\phi(M, M) \wedge \langle t, M\rangle \xrightarrow{\lambda} \langle t', M'\rangle \wedge \langle t', M'\rangle{\downarrow^{\phi}}M''$$

Then, by (*) follows $\langle t, M\rangle{\downarrow^{\phi}}M''$, hence $M'' \in \mathcal{O}^{\phi}(t, M)$.

□

We show that strong ϕ-refinement is sound with respect to the output function.

Lemma 5.4.7 *Let $\phi \subseteq \mathbb{M} \times \mathbb{M}$ be an interference set.*
If $\langle s, M\rangle \leq^{\phi} \langle t, M\rangle$, then $\mathcal{O}^{\phi}(s, M) \subseteq \mathcal{O}^{\phi}(t, M)$.

Proof Follows from Theorem 5.4.4. □

5.5 Weak Generic Refinement

Analogous to strong generic refinement, we develop in this section the theory of *weak* generic refinement which is indifferent to ε-transitions. As before, we use a binary relation ϕ over multisets to denote the possible interference from the environment.

Definition 5.5.1 *Let $\mathcal{R} \subseteq \mathbb{C} \times \mathbb{C}$ and $\phi \subseteq \mathbb{M} \times \mathbb{M}$.*
We say that $\mathcal{R}$ is a weak ϕ-simulation *if for all $(\langle s, M\rangle, \langle t, N\rangle) \in \mathcal{R}$, for all λ, for all M' such that $(M, M') \in \phi$*

1. $M = N$
2. $\langle s, M'\rangle \xrightarrow{\lambda} \langle s', M''\rangle \Rightarrow \exists t' : \langle t, M'\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle t', M''\rangle$ *such that* $(\langle s', M''\rangle, \langle t', M''\rangle) \in \mathcal{R}$ *and* $\overline{\lambda'} = \varepsilon^{k} \cdot \widehat{\lambda}$ *for some* $k \geq 0$
3. $s \equiv \mathsf{skip} \;\Rightarrow \langle t, M'\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle \mathsf{skip}, M'\rangle$ *where* $\widehat{\lambda'} = \langle\,\rangle$

We start by proving some basic properties of weak ϕ-simulation. We briefly postpone proving transitivity of weak ϕ-simulation because, in contrast to the strong variant, transitivity of weak ϕ-refinement requires an additional condition on ϕ.

Lemma 5.5.2 *If $\mathcal{R}_i$ are weak ϕ-simulations, then*

1. $Id_{\mathbb{C}} = \{(\langle s, M\rangle, \langle s, M\rangle) \mid \langle s, M\rangle \in \mathbb{C}\}$ *is a weak* ϕ*-simulation,*

2. $\bigcup_{i\in I} \mathcal{R}_i$ *is a weak* ϕ*-simulation.*

Proof

1. We verify the conditions of Definition 5.5.1. Let $(M, M') \in \phi$.

   1. Follows by reflexivity of $=$.

   2. From $\langle s, M'\rangle \xrightarrow{\lambda} \langle s', M''\rangle$ and $\longrightarrow \subseteq \longrightarrow^*$ follows $\langle s, M'\rangle \xrightarrow{\lambda}{}^* \langle s', M''\rangle$. If $\lambda = \varepsilon$, then $\overline{\lambda} = \varepsilon^k \cdot \widehat{\lambda}$ for $k = 1$. Otherwise, if $\lambda = \sigma$, then $\overline{\lambda} = \varepsilon^k \cdot \widehat{\lambda}$ for $k = 0$. By definition of $Id_{\mathbb{C}}$ follows $(\langle s', M''\rangle, \langle s', M''\rangle) \in Id_{\mathbb{C}}$.

   3. By reflexivity of $\longrightarrow^*$ follows $\langle \mathsf{skip}, M'\rangle \xrightarrow{\overline{\lambda}}{}^* \langle \mathsf{skip}, M'\rangle$ where $\overline{\lambda} = \langle\ \rangle$.

2. Let $\mathcal{R} = \bigcup_{i\in I} \mathcal{R}_i$. Suppose $(\langle s_1, M\rangle, \langle s_2, N\rangle) \in \mathcal{R}$ and $(M, M') \in \phi$.
   Then $(\langle s_1, M\rangle, \langle s_2, N\rangle) \in \mathcal{R}_i$ for some $i \in I$.
   We verify the conditions of Definition 5.5.1:

   1. Because $\mathcal{R}_i$ is a weak ϕ-simulation, we have $M = N$.

   2. If $\langle s_1, M'\rangle \xrightarrow{\lambda} \langle s_1', M''\rangle$, then $\langle s_2, M'\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle s_2', M''\rangle$ where $\overline{\lambda'} = \varepsilon^k \cdot \widehat{\lambda}$ for some $k \geq 0$ and $(\langle s_1', M''\rangle, \langle s_2', M''\rangle) \in \mathcal{R}_i$. From $\mathcal{R}_i \subseteq \mathcal{R}$ follows $(\langle s_1', M''\rangle, \langle s_2', M''\rangle) \in \mathcal{R}$.

   3. The case $s_1 \equiv \mathsf{skip}$ is analogously to case 2.

□

Weak ϕ-simulation is not in general transitive. We proceed by showing that transitivity can be obtained by the additional condition of reflexivity of ϕ. The fact that the weak notion of ϕ-simulation requires this additional property can be explained as follows.

Weak refinement equates the behaviour of $\langle s, M\rangle$ and $\langle t, M\rangle$ if every transition by either configuration may be matched by a sequence of transitions by the other which has the same effect on the multiset (either s or t may perform more ε-transitions than the other).

The clause $\langle t, M'\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle t', M''\rangle$ in Definition 5.5.1 of weak simulation implicitly assumes that t may make a sequence of transitions that is not influenced by interference.

The refining configuration s can only achieve the same effect on the multiset if the environment abstains from interfering whilst s is performing one *or more* transitions to match the behaviour of t. The possibility of non-interference is modelled by including the identity relation on multisets in the interference set.

Lemma 5.5.3 proves that if configurations $\langle s, M\rangle$ and $\langle t, M\rangle$ are related by a weak interference closed ϕ-simulation where ϕ is reflexive, then t can simulate (in a "weak" fashion) any *sequence* of transitions that s can make.

Lemma 5.5.3 *Let ϕ be reflexive. Let $\mathcal{R}$ be an interference closed weak ϕ-simulation. If $(\langle s, M\rangle, \langle t, M\rangle) \in \mathcal{R}$, $\phi(M, M')$, and $\langle s, M'\rangle \xrightarrow{\overline{\lambda}}{}^* \langle s', M''\rangle$ where $\overline{\lambda} = \langle\, \lambda_1, \ldots, \lambda_n \,\rangle$, then $\langle t, M'\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle t', M''\rangle$ such that $(\langle s', M''\rangle, \langle t', M''\rangle) \in \mathcal{R}$ and $\overline{\lambda'} = \varepsilon^{k_1} \cdot \widehat{\lambda_1} \cdot \ldots \cdot \varepsilon^{k_n} \cdot \widehat{\lambda_n}$ where $k_i \geq 0$ for all $i : 1 \leq i \leq n$.*

Proof By induction on the length n of the transition sequence $\overline{\lambda}$.

- $n = 0$: Then $s' = s$, $M'' = M'$ and $\overline{\lambda} = \langle\, \rangle$.
  By definition of $\longrightarrow^*$ follows $\langle t, M'\rangle \xrightarrow{\langle\, \rangle}{}^* \langle t, M'\rangle$.
  Since $\mathcal{R}$ is interference closed, it follows that $(\langle s, M'\rangle, \langle t, M'\rangle) \in \mathcal{R}$.

- $n > 0$: The transition sequence can be written $\langle s, M'\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle s'', M'''\rangle \xrightarrow{\lambda_n} \langle s', M''\rangle$ where $\overline{\lambda} = \overline{\lambda'} \cdot \lambda_n$. The induction hypothesis gives $\langle t, M'\rangle \xrightarrow{\overline{\mu'}}{}^* \langle t'', M'''\rangle$ such that $(\langle s'', M'''\rangle, \langle t'', M'''\rangle) \in \mathcal{R}$ and $\overline{\mu'} = \varepsilon^{k_1} \cdot \widehat{\lambda_1} \cdot \ldots \cdot \varepsilon^{k_{n-1}} \cdot \widehat{\lambda_{n-1}}$ where $k_i \geq 0$ for all $i : 1 \leq i \leq n-1$. From $\langle s'', M'''\rangle \xrightarrow{\lambda_n} \langle s', M''\rangle$ follows, by $(\langle s'', M'''\rangle, \langle t'', M'''\rangle) \in \mathcal{R}$ and $\phi(M''', M''')$, that $\langle t'', M'''\rangle \xrightarrow{\overline{\mu''}}{}^* \langle t', M''\rangle$ such that $(\langle s', M''\rangle, \langle t', M''\rangle) \in \mathcal{R}$ and $\overline{\mu''} = \varepsilon^{k_n} \cdot \widehat{\lambda_n}$. Hence $\widehat{\mu' \cdot \mu''} = \varepsilon^{k_1} \cdot \widehat{\lambda_1} \cdot \ldots \cdot \varepsilon^{k_n} \cdot \widehat{\lambda_n}$ where $k_i \geq 0$ for all $i : 1 \leq i \leq n$.

□

Lemma 5.5.4 *Let $\phi \subseteq M \times M$ be reflexive. Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be weak ϕ-simulations. If $\mathcal{R}_2$ is interference closed, then $\mathcal{R}_1\mathcal{R}_2$ is a weak ϕ-simulation.*

Proof Let $\mathcal{R} = \mathcal{R}_1\mathcal{R}_2$. Suppose $(\langle s_1, M\rangle, \langle s_2, N\rangle) \in \mathcal{R}$ and $(M, M') \in \phi$.
Then for some t and N' we have $(\langle s_1, M\rangle, \langle t, N'\rangle) \in \mathcal{R}_1$ and $(\langle t, N'\rangle, \langle s_2, N\rangle) \in \mathcal{R}_2$.
Because $\mathcal{R}_1$ and $\mathcal{R}_2$ are weak ϕ-simulations, $M = N' = N$.
transition
Assume $\langle s_1, M'\rangle \xrightarrow{\lambda} \langle s_1', M''\rangle$. Then from $(\langle s_1, M\rangle, \langle t, M\rangle) \in \mathcal{R}_1$ and $\phi(M, M')$ follows $\langle t, M'\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle t', M''\rangle$ such that $(\langle s_1', M''\rangle, \langle t', M''\rangle) \in \mathcal{R}_1$ where $\overline{\lambda'} = \varepsilon^k \cdot \widehat{\lambda}$ for some

$k \geq 0$. From $(\langle t, M\rangle, \langle s_2, M\rangle) \in \mathcal{R}_2$, $\phi(M, M')$ and reflexivity of ϕ follows by Lemma 5.5.3 that $\langle s_2, M'\rangle \xrightarrow{\overline{\lambda''}}{}^* \langle s_2', M''\rangle$ such that $(\langle t', M''\rangle, \langle s_2', M''\rangle) \in \mathcal{R}_2$ and $\overline{\lambda''} = \varepsilon^{k'} \cdot \widehat{\lambda}$ for some $k' \geq 0$. From $(\langle s_1', M''\rangle, \langle t', M''\rangle) \in \mathcal{R}_1$ and $(\langle t', M''\rangle, \langle s_2', M''\rangle) \in \mathcal{R}_2$ follows $(\langle s_1', M''\rangle, \langle s_2', M''\rangle) \in \mathcal{R}$.
termination
If $s_1 \equiv \mathsf{skip}$ then, from $(\langle s_1, M\rangle, \langle t, M\rangle) \in \mathcal{R}_1$, we have $\langle t, M'\rangle \xrightarrow{\overline{\lambda}}{}^* \langle \mathsf{skip}, M'\rangle$ where $\widehat{\lambda} = \langle\,\rangle$. From $(\langle t, M\rangle, \langle s_2, M\rangle) \in \mathcal{R}_2$ and reflexivity of ϕ follows by Lemma 5.5.3 that $\langle s_2, M'\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle \mathsf{skip}, M'\rangle$ where $\widehat{\lambda'} = \langle\,\rangle$. □

Lemma 5.5.5 shows that interference closed weak ϕ-simulations are transitive, provided that ϕ is reflexive.

Lemma 5.5.5 *Let $\phi \subseteq \mathbb{M} \times \mathbb{M}$ be reflexive. If $\mathcal{R}_1$ and $\mathcal{R}_2$ are interference closed weak ϕ-simulations, then $\mathcal{R}_1\mathcal{R}_2$ is an interference closed weak ϕ-simulation.*

Proof From reflexivity of ϕ follows by Lemma 5.5.4 that $\mathcal{R}_1\mathcal{R}_2$ is a weak ϕ-simulation. It remains to show that $\mathcal{R}_1\mathcal{R}_2$ is interference closed.

Assume $\langle s, M\rangle \mathcal{R}_1\mathcal{R}_2 \langle s', M\rangle$ and $\phi(M, M')$. Then $\langle s, M\rangle \mathcal{R}_1 \langle t, M\rangle$ and $\langle t, M\rangle \mathcal{R}_2 \langle s', M\rangle$ for some t. Because $\mathcal{R}_1$ and $\mathcal{R}_2$ are interference closed, we get by Definition 5.3.5, that $\langle s, M'\rangle \mathcal{R}_1 \langle t, M'\rangle$ and $\langle t, M'\rangle \mathcal{R}_2 \langle s', M'\rangle$. Hence $\langle s, M'\rangle \mathcal{R}_1\mathcal{R}_2 \langle s', M'\rangle$. □

Next, we show that if ϕ is transitive, then a weak ϕ-simulation is interference closed.

Lemma 5.5.6 *Let $\mathcal{R}$ be a weak ϕ-simulation. If ϕ is transitive, then $\mathcal{R}$ is interference closed.*

Proof We need to show that if $s\mathcal{R}t$ and $\phi(M, M')$, then $s\mathcal{R}M't$.
Suppose $\phi(M', M'')$. By transitivity of ϕ follows $\phi(M, M'')$.
transition
If $\langle s, M''\rangle \xrightarrow{\lambda} \langle s', M'''\rangle$, then by $\langle s, M\rangle \mathcal{R} \langle t, M\rangle$ and $\phi(M, M'')$ follows $\langle t, M''\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle t', M'''\rangle$ such that $s'\mathcal{R}M'''t'$ and $\overline{\lambda'} = \varepsilon^k \cdot \widehat{\lambda}$ for some $k \geq 0$.
termination
If $s \equiv \mathsf{skip}$ then by $\langle s, M\rangle \mathcal{R} \langle t, M\rangle$ and $\phi(M, M'')$ follows $\langle t, M''\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle \mathsf{skip}, M''\rangle$ where $\widehat{\lambda'} = \langle\,\rangle$. □

Now that the necessary basic properties of weak ϕ-simulation have been establishes, we proceed by defining weak ϕ-refinement, denoted $\lesssim^{\phi}$, as the maximal weak

ϕ-simulation. Weak ϕ-equivalence, denoted $\simeq^{\phi}$, is defined as the intersection of $\lesssim^{\phi}$ and its inverse.

Definition 5.5.7

1. $\lesssim^{\phi} = \bigcup\{\mathcal{R} \mid \mathcal{R}$ *is a weak* ϕ*-simulation* $\}$
2. $\simeq^{\phi} = \lesssim^{\phi} \cap \lesssim^{\phi\,-1}$
3. $s \lesssim^{\phi}_{M} t$ *iff* $\langle s, M\rangle \lesssim^{\phi} \langle t, M\rangle$

Next, we show that $\lesssim^{\phi}$ is interference closed if ϕ is transitive.

Corollary 5.5.8 *If* ϕ *is transitive, then* $\lesssim^{\phi}$ *is interference closed*

Proof Follows from Lemma 5.5.6 because $\lesssim^{\phi}$ is a weak ϕ-simulation and ϕ is transitive. □

Lemma 5.5.9

1. $\lesssim^{\phi}$ *is the largest weak* ϕ*-simulation.*
2. *If* ϕ *is reflexive and transitive, then* $\lesssim^{\phi}$ *is a partial order.*
3. *If* ϕ *is reflexive and transitive, then* $\simeq^{\phi}$ *is an equivalence relation.*

Proof

1. By Lemma 5.5.2.1 $\lesssim^{\phi}$ is a weak ϕ-simulation and by Definition 5.5.7.1 it includes any other such.
2. We consider the following properties:
   - Reflexivity: follows from Lemma 5.5.2.1.
   - Transitivity: from transitivity of ϕ follows by Lemma 5.5.8 that $\lesssim^{\phi}$ is interference closed. Furthermore, $\lesssim^{\phi}$ is a weak ϕ-simulation, hence by reflexivity of ϕ and Lemma 5.5.5 follows that the composition $\lesssim^{\phi} \lesssim^{\phi}$ is a weak ϕ-simulation that is interference closed. By Lemma 5.5.9.1 follows $\lesssim^{\phi} \lesssim^{\phi} \subseteq \lesssim^{\phi}$.
   - Antisymmetry: follows from Lemma 5.5.9.3.
3. We consider the following properties:

- Reflexivity: follow from Lemma 5.5.9.2 and Definition 5.5.7.2.
- Transitivity: follows from transitivity of $\lesssim^{\phi}$ (Lemma 5.5.9.2) and Definition 5.5.7.2.
- Symmetry: follows from Definition 5.5.7.2.

□

We use some fixed-point theory to show that $\lesssim^{\phi}$ defines the relation that contains precisely all weak ϕ-simulations.

Definition 5.5.10 *Define a function* $\mathbb{F} : \mathbb{C} \times \mathbb{C} \to \mathbb{C} \times \mathbb{C}$ *as follows:*
If $\mathcal{R} \subseteq \mathbb{C} \times \mathbb{C}$, *then* $(\langle s, M\rangle, \langle t, M\rangle) \in \mathbb{F}(\mathcal{R})$ *iff, for all* λ, *for all* $M' : \phi(M, M')$,

1. $M = N$
2. $\langle s, M'\rangle \xrightarrow{\lambda} \langle s', M''\rangle \Rightarrow \exists t' : \langle t, M'\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle t', M''\rangle$ *such that* $(\langle s', M''\rangle, \langle t', M''\rangle) \in \mathcal{R}$ *and* $\overline{\lambda'} = \varepsilon^{k} \cdot \widehat{\lambda}$ *for some* $k \geq 0$
3. $s \equiv \mathsf{skip} \Rightarrow \langle t, M'\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle \mathsf{skip}, M'\rangle$ *where* $\widehat{\lambda} = \langle\ \rangle$

Theorem 5.5.11 $\lesssim^{\phi}$ *is largest fixed point of* $\mathbb{F}$.

Proof Analogous to the proof of Theorem 5.2.7. □

The theory of *up-to* simulations is developed next for weak ϕ-simulation.

Definition 5.5.12 *Let* $\mathcal{R} \subseteq \mathbb{C} \times \mathbb{C}$ *and* $\phi \subseteq \mathbb{M} \times \mathbb{M}$.
We say that $\mathcal{R}$ *is a* weak ϕ-simulation up-to $\lesssim^{\phi}$ *if for all* $(\langle s, M\rangle, \langle t, N\rangle) \in \mathcal{R}$, *for all* λ, *for all* $M' : (M, M') \in \phi$,

1. $M = N$
2. $\langle s, M'\rangle \xrightarrow{\lambda} \langle s', M''\rangle \Rightarrow \exists t' : \langle t, M'\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle t', M''\rangle$ *such that* $(\langle s', M''\rangle, \langle t', M''\rangle) \in \lesssim^{\phi} \mathcal{R} \lesssim^{\phi}$ *and* $\overline{\lambda'} = \varepsilon^{k} \cdot \widehat{\lambda}$ *for some* $k \geq 0$
3. $s \equiv \mathsf{skip} \Rightarrow \langle t, M'\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle \mathsf{skip}, M\rangle$ *where* $\widehat{\lambda} = \langle\ \rangle$

Lemma 5.5.13 *Let* $\phi \subseteq \mathbb{M} \times \mathbb{M}$ *be reflexive. Let* $\mathcal{R}$ *be a weak* ϕ*-simulation up-to* $\lesssim^{\phi}$ *that is interference closed. If* $(\langle s, M\rangle, \langle t, M\rangle) \in \mathcal{R}$, $\phi(M, M')$, *and* $\langle s, M'\rangle \xrightarrow{\overline{\lambda}}{}^{*} \langle s', M''\rangle$ *where* $\overline{\lambda} = \langle \lambda_1, \ldots, \lambda_n \rangle$, *then* $\langle t, M'\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle t', M''\rangle$ *such that* $(\langle s', M''\rangle, \langle t', M''\rangle) \in \lesssim^{\phi} \mathcal{R} \lesssim^{\phi}$ *where* $\overline{\lambda'} = \varepsilon^{k_1} \cdot \widehat{\lambda_1} \cdot \ldots \cdot \varepsilon^{k_n} \cdot \widehat{\lambda_n}$ *with* $k_i \geq 0$ *for all* $i : 1 \leq i \leq n$.

Proof The proof proceeds analogously to the proof of Lemma 5.5.3. □

Lemma 5.5.14
Let $\phi \subseteq \mathbb{M} \times \mathbb{M}$ be reflexive and transitive. Let $\mathcal{R}$ be a weak ϕ-simulation up-to $\lesssim^\phi$.
If $\mathcal{R}$ is interference closed, then $\lesssim^\phi \mathcal{R} \lesssim^\phi$ is a weak ϕ-simulation.

Proof
transition
Assume $\langle s, M\rangle \lesssim^\phi \mathcal{R} \lesssim^\phi \langle t, M\rangle$ and $\phi(M, M')$. Hence $\langle s, M\rangle \lesssim^\phi \langle s_1, M\rangle$, $\langle s_1, M\rangle \mathcal{R} \langle t_1, M\rangle$ and $\langle t_1, M\rangle \lesssim^\phi \langle t, M\rangle$ for some s_1 and t_1. Assume $\langle s, M'\rangle \xrightarrow{\lambda} \langle s', M''\rangle$.
From $\langle s, M\rangle \lesssim^\phi \langle s_1, M\rangle$ follows $\langle s_1, M'\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle s_1', M''\rangle$ such that $\langle s', M''\rangle \lesssim^\phi \langle s_1', M''\rangle$ and $\overline{\lambda} = \varepsilon^k \cdot \widehat{\lambda'}$ for some $k \geq 0$. From $\langle s_1, M\rangle \mathcal{R} \langle t_1, M\rangle$ and Lemma 5.5.13 follows $\langle t_1, M'\rangle \xrightarrow{\overline{\lambda''}}{}^* \langle t_1', M''\rangle$ such that $\langle s_1', M''\rangle \lesssim^\phi \mathcal{R} \lesssim^\phi \langle t_1', M''\rangle$ and $\overline{\lambda''} = \varepsilon^{k'} \cdot \widehat{\lambda}$ for some $k' \geq 0$. From $\langle t_1, M\rangle \lesssim^\phi \langle t, M\rangle$ and Lemma 5.5.3 follows $\langle t, M'\rangle \xrightarrow{\overline{\lambda'''}}{}^* \langle t', M''\rangle$ such that $\langle t_1', M''\rangle \lesssim^\phi \langle t', M''\rangle$ and $\overline{\lambda'''} = \varepsilon^{k''} \cdot \widehat{\lambda}$ for some $k'' \geq 0$.
Hence, $\langle s', M''\rangle \lesssim^\phi \lesssim^\phi \mathcal{R} \lesssim^\phi \lesssim^\phi \langle t', M''\rangle$. By transitivity of ϕ and Lemma 5.5.9.2 follows transitivity of $\lesssim^\phi$, hence $\langle s', M''\rangle \lesssim^\phi \mathcal{R} \lesssim^\phi \langle t', M''\rangle$.

termination: Follows analogously. □

Lemma 5.5.15 *Let $\phi \subseteq \mathbb{M} \times \mathbb{M}$ be reflexive and transitive. Let $\mathcal{R}$ be a weak ϕ-simulation up-to $\lesssim^\phi$.*
If $\mathcal{R}$ is interference closed, then $\mathcal{R} \subseteq \lesssim^\phi$.

Proof From Lemma 5.5.14 and Lemma 5.5.9.1 follows $\lesssim^\phi \mathcal{R} \lesssim^\phi \subseteq \lesssim^\phi$.
From $Id_{\mathbb{C}} \subseteq \lesssim^\phi$ follows $\mathcal{R} \subseteq \lesssim^\phi$. □

The notions of weak statebased and stateless refinement fit into the framework of ϕ-refinement analogously to Theorems 5.2.11 and 5.2.12. In order use the generic theory of refinement to prove that the weak notions of refinement satisfy the properties proposed in previous sections, we need to check that the interference parameter satisfies certain properties.

Recall that statebased refinement is obtained from ϕ-refinement by taking $\phi_{statebased} = Id_{\mathbb{M}}$. Hence $\phi_{statebased}$ is reflexive and transitive. The properties ascribed to weak statebased simulation and refinement, in Propositions 4.3.15 and 4.3.17, follow from

Lemmas 5.5.2 and 5.5.9. Theorem 5.5.11 proves that $\precapprox$ is the largest relation that satisfies the definition of weak statebased simulation. Hence $\precapprox$ defines the relation that contains precisely all weak statebased simulations. The justification of the up-to method for weak statebased refinement, suggested in Proposition 4.3.19, follows by Lemma 5.5.15.

Next, we consider the weak stateless variant. This is obtained from ϕ-refinement by taking $\phi_{stateless} = \mathbb{M} \times \mathbb{M}$. Hence $\phi_{stateless}$ is reflexive and transitive. The properties ascribed to weak stateless refinement in Proposition 4.4.31 follow from Lemma 5.5.9. Theorem 5.5.11 proves that $\precsim$ is the largest relation that satisfies the definition of weak stateless simulation. Hence $\precsim$ defines the relation that contains precisely all weak stateless simulations. The justification of the up-to method for weak stateless refinement, suggested in Proposition 4.4.33, follows by Lemma 5.5.15.

The weak variants of ϕ-refinement are, just as the strong variants, inversely ordered by subset inclusion of the interference set.

Theorem 5.5.16 *Let $\phi, \psi \subseteq \mathbb{M} \times \mathbb{M}$. If $\phi \subseteq \psi$ then $\lesssim^{\psi} \subseteq \lesssim^{\phi}$.*

Proof Analogous to the proof of Theorem 5.2.13. □

Consequently, weak stateless refinement is contained in weak statebased refinement.

Corollary 5.5.17 *If $s \precsim t$, then $\langle s, M\rangle \precapprox \langle t, M\rangle$ for all $M \in \mathbb{M}$.*

Proof By Theorem 5.5.16 from $Id_{\mathbb{M}} \subset \mathbb{M} \times \mathbb{M}$. □

Furthermore, strong ϕ-refinement is contained in weak ϕ-refinement.

Theorem 5.5.18 *For all ϕ : $\leq^{\phi} \subseteq \lesssim^{\phi}$.*

Proof Straightforward from the definitions of strong ϕ-refinement and weak ϕ-refinement. □

Consequently, strong statebased refinement and strong stateless refinement are contained in weak statebased refinement and weak stateless refinement respectively.

Corollary 5.5.19

1. *If $\langle s, M\rangle \leqq \langle t, M\rangle$, then $\langle s, M\rangle \precapprox \langle t, M\rangle$ (or: $\leqq \subseteq \precapprox$).*

2. If $s \leqslant t$, then $s \precsim t$ (or: $\leqslant \subseteq \precsim$).

Proof By Theorem 5.5.18. □

5.6 Precongruence of Weak Generic Refinement

In this section we show under which conditions on the interference set ϕ the corresponding notion of weak generic refinement is a precongruence. To this end we show that weak generic refinement is preserved by the combinators from the coordination language.

In the remainder of this section, we assume that the schedules are taken from a transition closed set $\mathbb{S}'$ of schedules (i.e. $\mathbb{S}'$ satisfies condition $(S1)$ from Definition 5.3.1). Furthermore we assume that, as was the case for precongruence of strong generic refinement, that ϕ satisfies $(P1)$ and $(P2)$ (from Definition 5.3.4). From $(P2)$ follows that ϕ is transitive. For the lemmas in this section, we additionally require that ϕ is reflexive.

(P3) For all M, $\phi(M, M)$ (reflexivity).

First, we show that structural equivalence implies weak ϕ-equivalence.

Lemma 5.6.1 *Let $\mathbb{S}'$ satisfy $(S1)$ and let ϕ satisfy $(P1), (P2)$ and $(P3)$. Let $s, t \in \mathbb{S}'$. If $s \equiv t$, then for all M, $s \simeq^{\phi}_{M} t$.*

Proof Using (N8) it is straightforward to prove $s \lesssim^{\phi}_{M} t$ and $t \lesssim^{\phi}_{M} s$. □

We proceed by showing that all combinators from our coordination language respect the ordering $\lesssim^{\phi}_{M}$.

Lemma 5.6.2 *Let $\mathbb{S}'$ satisfy $(S1)$ and let ϕ satisfy $(P1), (P2)$ and $(P3)$.*
Let $s_1, s_2, t_1, t_2 \in \mathbb{S}'$. If $s_1 \lesssim^{\phi}_{M} s_2$ and $t_1 \lesssim^{\phi}_{M} t_2$, then $r \to s_1[t_1] \lesssim^{\phi}_{M} r \to s_2[t_2]$.

Proof
Assume $\phi(M, M')$ and consider the following cases:
transition

- Assume $\langle r \to s_1[t_1], M' \rangle \xrightarrow{\varepsilon} \langle t_1, M' \rangle$. Then by (N0) $\langle r \to s_2[t_2], M' \rangle \xrightarrow{\varepsilon} \langle t_2, M' \rangle$. By definition of $\longrightarrow^*$ follows $\langle r \to s_2[t_2], M' \rangle \xrightarrow{\varepsilon}{}^* \langle t_2, M' \rangle$. Clearly $\varepsilon = \varepsilon \cdot \widehat{\varepsilon}$. From $t_1 \lesssim^{\phi}_{M} t_2$ and $\phi(M, M')$ follows, by Lemma 5.5.8, that $t_1 \lesssim^{\phi}_{M'} t_2$.

- Assume $\langle r \to s_1[t_1], M'\rangle \xrightarrow{\sigma} \langle s_1, M''\rangle$. Then by (N1) $\langle r \to s_2[t_2], M'\rangle \xrightarrow{\sigma} \langle s_2, M''\rangle$. By definition of $\longrightarrow^*$ follows $\langle r \to s_2[t_2], M'\rangle \xrightarrow{\sigma}{}^* \langle s_2, M''\rangle$. Clearly $\sigma = \varepsilon^0 \cdot \sigma$. By $(P1)$ follows $\phi(M', M'')$. Then, by $(P2)$, follows $\phi(M, M'')$. From $s_1 \lesssim^\phi_M s_2$ and Lemma 5.5.8 follows $s_1 \lesssim^\phi_{M''} s_2$.

termination
Holds vacuously. □

Lemma 5.6.3 *Let $\mathbb{S}'$ satisfy $(S1)$ and let ϕ satisfy $(P1), (P2)$ and $(P3)$. Let $s_1, s_2, t_1, t_2 \in \mathbb{S}'$. If $s_1 \lesssim^\phi_M s_2$ and $t_1 \lesssim^\phi_M t_2$, then $s_1; t_1 \lesssim^\phi_M s_2; t_2$.*

Proof Let $\mathcal{R} = \{(\langle s_1; t_1, M\rangle, \langle s_2; t_2, M\rangle) \mid s_1 \lesssim^\phi_M s_2, t_1 \lesssim^\phi_M t_2\}$.
From transitivity of ϕ (by $(P2)$) and Lemma 5.5.6 follows that $\mathcal{R}$ is interference closed. The result follows from Lemma 5.5.14 by showing that $\mathcal{R}$ is a weak ϕ-simulation *up-to* $\lesssim^\phi$.
Assume $\phi(M, M')$.
transition
Suppose $\langle s_1; t_1, M'\rangle \xrightarrow{\lambda} \langle s_1'; t_1', M''\rangle$. Then by $(P1)$ follows $\phi(M', M'')$ and by $(P2)$ follows $\phi(M, M'')$.
Consider the possible derivations of this transition

- By (N5) from $\langle s_1, M'\rangle \xrightarrow{\lambda} \langle s_1', M''\rangle$. From $s_1 \lesssim^\phi_M s_2$ and $\phi(M, M')$ follows $\langle s_2, M'\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle s_2', M''\rangle$ such that $s_1' \lesssim^\phi_{M''} s_2'$ and $\overline{\lambda'} = \varepsilon^k \cdot \widehat{\lambda}$ for some $k \geq 0$. Using (N5) we derive $\langle s_2; t_2, M'\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle s_2'; t_2, M''\rangle$. From $t_1 \lesssim^\phi_M t_2$, $\phi(M, M'')$ and Lemma 5.5.8 follows $t_1 \lesssim^\phi_{M''} t_2$. By $Id_{\mathbb{C}} \subseteq \lesssim^\phi$ follows $(\langle s_1'; t_1, M''\rangle, \langle s_2'; t_2, M''\rangle) \in \lesssim^\phi \mathcal{R} \lesssim^\phi$.

- By (N8) from $s_1 \equiv \mathsf{skip}$ and $\langle t_1, M'\rangle \xrightarrow{\lambda} \langle t_1', M''\rangle$. From $s_1 \lesssim^\phi_M s_2$ follows $\langle s_2, M'\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle \mathsf{skip}, M'\rangle$ where $\widehat{\lambda'} = \langle\,\rangle$, hence $\overline{\lambda'} = \varepsilon^k$ for some $k \geq 0$. From $t_1 \lesssim^\phi_M t_2$ follows $\langle t_2, M'\rangle \xrightarrow{\overline{\lambda''}}{}^* \langle t_2', M''\rangle$ such that $t_1' \lesssim^\phi_{M''} t_2'$ and $\overline{\lambda''} = \varepsilon^{k'} \cdot \widehat{\lambda}$ for some $k' \geq 0$. By (N5) and (N8) we derive

$$\langle s_2; t_2, M'\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle t_2, M'\rangle \xrightarrow{\overline{\lambda''}}{}^* \langle t_2', M''\rangle$$

  hence $\langle s_2; t_2, M'\rangle \xrightarrow{\overline{\lambda'}\cdot\overline{\lambda''}}{}^* \langle t_2', M''\rangle$ where $\overline{\lambda' \cdot \lambda''} = \varepsilon^{k+k'} \cdot \widehat{\lambda}$ and $k + k' \geq 0$.
  By Lemma 5.6.1 follows $t_1' \lesssim^\phi_{M''} \mathsf{skip}; t_1'$ and $\mathsf{skip}; t_2' \lesssim^\phi_{M''} t_2'$. Hence from $(\mathsf{skip}; t_1', M'', \mathsf{skip}; t_2', M'') \in \mathcal{R}$ follows $(\langle t_1', M''\rangle, \langle t_2', M''\rangle) \in \lesssim^\phi \mathcal{R} \lesssim^\phi$.

termination
$s_1; t_1 \equiv \mathsf{skip}$ only if $s_1 \equiv \mathsf{skip}$ and $t_1 \equiv \mathsf{skip}$. From $s_1 \lesssim^{\phi}_{M} s_2$ and $t_1 \lesssim^{\phi}_{M} t_2$ follows $\langle s_2, M' \rangle \xrightarrow{\overline{\lambda_1}}{}^{*} \langle \mathsf{skip}, M' \rangle$ and $\langle t_2, M' \rangle \xrightarrow{\overline{\lambda_2}}{}^{*} \langle \mathsf{skip}, M' \rangle$ where $\widehat{\lambda_1} = \langle\,\rangle$ and $\widehat{\lambda_2} = \langle\,\rangle$. By (N5) and (N8) follows $\langle s_2; t_2, M' \rangle \xrightarrow{\overline{\lambda_1 \cdot \lambda_2}}{}^{*} \langle \mathsf{skip}, M' \rangle$ where $\widehat{\lambda_1 \cdot \lambda_2} = \langle\,\rangle$. □

Lemma 5.6.4 *Let $\mathbb{S}'$ satisfy $(S1)$ and let ϕ satisfy $(P1), (P2)$ and $(P3)$.*
let $s_1, s_2, t_1, t_2 \in \mathbb{S}'$. If $s_1 \lesssim^{\phi}_{M} s_2$ and $t_1 \lesssim^{\phi}_{M} t_2$, then $s_1 \parallel t_1 \lesssim^{\phi}_{M} s_2 \parallel t_2$.

Proof Let $\mathcal{R} = \{(\langle s_1 \parallel t_1, M \rangle, \langle s_2 \parallel t_2, M \rangle) \mid s_1 \lesssim^{\phi}_{M} s_2, t_1 \lesssim^{\phi}_{M} t_2\}$.
We show that $\mathcal{R}$ is a weak ϕ-simulation by transition induction. Assume $\phi(M, M')$.
transition
Assume $\langle s_1 \parallel t_1, M' \rangle \xrightarrow{\lambda} \langle s_1' \parallel t_1', M'' \rangle$. By $(P1)$ follows $\phi(M', M'')$. Then by $(P2)$ follows $\phi(M, M'')$. Consider the different ways in which the last inference can be made.

1. By (N2) from $\langle s_1, M' \rangle \xrightarrow{\lambda} \langle s_1', M'' \rangle$. From $s_1 \lesssim^{\phi}_{M} s_2$ follows $\langle s_2, M' \rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle s_2', M'' \rangle$ such that $s_1' \lesssim^{\phi}_{M''} s_2'$ and $\overline{\lambda'} = \varepsilon^k \cdot \widehat{\lambda}$ for some $k \geq 0$. Using (N2) we derive $\langle s_2 \parallel t_2, M' \rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle s_2' \parallel t_2, M'' \rangle$. By Lemma 5.5.8 we get from $t_1 \lesssim^{\phi}_{M} t_2$ and $\phi(M, M'')$ that $t_1 \lesssim^{\phi}_{M''} t_2$. Hence $(\langle s_1' \parallel t_1, M'' \rangle, \langle s_2' \parallel t_2, M'' \rangle) \in \mathcal{R}$.

2. By (N2) from $\langle t_1, M' \rangle \xrightarrow{\lambda} \langle t_1', M'' \rangle$. The proof is analogous to the previous case.

3. By (N3) from $\langle s_1, M' \rangle \xrightarrow{\lambda} \langle s_1', M'' \rangle$ and $\langle t_1, M' \rangle \xrightarrow{\varepsilon} \langle t_1', M' \rangle$. From $s_1 \lesssim^{\phi}_{M} s_2$ follows $\langle s_2, M' \rangle \xrightarrow{\overline{\lambda_1}}{}^{*} \langle s_2', M'' \rangle$ such that $s_1' \lesssim^{\phi}_{M''} s_2'$ and $\overline{\lambda_1} = \varepsilon^{k_1} \cdot \widehat{\lambda}$ for some $k_1 \geq 0$. From $t_1 \lesssim^{\phi}_{M} t_2$ follows $\langle t_2, M' \rangle \xrightarrow{\overline{\lambda_2}}{}^{*} \langle t_2', M' \rangle$ such that $t_1' \lesssim^{\phi}_{M'} t_2'$ and $\overline{\lambda_2} = \varepsilon^{k_2} \cdot \widehat{\varepsilon}$ for some $k_2 \geq 0$. By (repeated use of) (N2) follows

$$\langle s_2 \parallel t_2, M' \rangle \xrightarrow{\overline{\lambda_2}}{}^{*} \langle s_2 \parallel t_2', M' \rangle \xrightarrow{\overline{\lambda_1}}{}^{*} \langle s_2' \parallel t_2', M'' \rangle$$

hence, by transitivity of $\longrightarrow^{*}$ follows $\langle s_2 \parallel t_2, M' \rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle s_2' \parallel t_2', M'' \rangle$ where $\overline{\lambda'} = \overline{\lambda_2} \cdot \overline{\lambda_1}$ hence $\overline{\lambda'} = \varepsilon^{k_2 + k_1} \cdot \widehat{\lambda}$. From $\phi(M', M'')$ and Lemma 5.5.8 follows $t_1' \lesssim^{\phi}_{M''} t_2'$. Thus $(\langle s_1' \parallel t_1', M'' \rangle, \langle s_2' \parallel t_2', M'' \rangle) \in \mathcal{R}$.

4. By (N3) from $\langle s_1, M' \rangle \xrightarrow{\varepsilon} \langle s_1', M' \rangle$ and $\langle t_1, M' \rangle \xrightarrow{\lambda} \langle t_1', M'' \rangle$.
The proof is analogous to the previous case.

5. By (N4) from $\langle s_1, M' \rangle \xrightarrow{\sigma_1} \langle s_1', M_1 \rangle$, $\langle t_1, M' \rangle \xrightarrow{\sigma_2} \langle t_1', M_2 \rangle$, and $M' \models \sigma_1 \bowtie \sigma_2$.
From $s_1 \lesssim^{\phi}_{M} s_2$ follows $\langle s_2, M' \rangle \xrightarrow{\overline{\lambda_1}}{}^{*} \langle s_2', M_1 \rangle$ such that $s_1' \lesssim^{\phi}_{M_1} s_2'$ and $\overline{\lambda_1} = \varepsilon^{k_1} \cdot \sigma_1$

for some $k_1 \geq 0$. From $t_1 \lesssim^{\phi}_{M} t_2$ follows $\langle t_2, M' \rangle \xrightarrow{\overline{\lambda_2}}{}^{*} \langle t_2', M_2 \rangle$ such that $t_1' \lesssim^{\phi}_{M_2} t_2'$ and $\overline{\lambda_2} = \varepsilon^{k_2} \cdot \sigma_2$ for some $k_2 \geq 0$. By definition of $\longrightarrow^{*}$, these transitions can also be written as

$$\langle s_2, M' \rangle \xrightarrow{\varepsilon^{k_1}}{}^{*} \langle s_2'', M' \rangle \xrightarrow{\sigma_1} \langle s_2', M_1 \rangle \tag{5.1}$$

and

$$\langle t_2, M' \rangle \xrightarrow{\varepsilon^{k_2}}{}^{*} \langle t_2'', M' \rangle \xrightarrow{\sigma_2} \langle t_2', M_2 \rangle \tag{5.2}$$

By ($k_1 + k_2$ times) (N2), then by (N4) (which is possible because $M' \models \sigma_1 \bowtie \sigma_2$) we derive

$$\langle s_2 \parallel t_2, M' \rangle \xrightarrow{\varepsilon^{k_1+k_2}}{}^{*} \langle s_2'' \parallel t_2'', M' \rangle \xrightarrow{\sigma_1 \cdot \sigma_2} \langle s_2' \parallel t_2', M'' \rangle$$

Hence

$$\langle s_2 \parallel t_2, M' \rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle s_2' \parallel t_2', M'' \rangle$$

where $\overline{\lambda'} = \varepsilon^{k_1+k_2} \cdot \sigma$ and $k_1 + k_2 \geq 0$.

It remains to show that $s_1' \lesssim^{\phi}_{M''} s_2'$ and $t_1' \lesssim^{\phi}_{M''} t_2'$. To this end, we first show that there exists transitions from M_1 and M_2 to M'': from $M' \models \sigma_1 \bowtie \sigma_2$, (5.1) and (5.2) follows by Lemma A.2.6 that the following transitions exist

$$\langle t_2'', M_1 \rangle \xrightarrow{\sigma_2} \langle t_2', M'' \rangle \tag{5.3}$$

$$\langle s_2'', M_2 \rangle \xrightarrow{\sigma_1} \langle s_2', M'' \rangle \tag{5.4}$$

Then, by $(P1)$ follows that if there is a transition from a multiset N to N', then $\phi(N, N')$. Hence, from (5.3) follows by $(P1)$ that $\phi(M_1, M'')$. Then, from the fact that $\lesssim^{\phi}$ is interference closed follows from $s_1' \lesssim^{\phi}_{M_1} s_2'$ and $\phi(M_1, M'')$ by Lemma 5.5.8 that $s_1' \lesssim^{\phi}_{M''} s_2'$.

Analogously, we infer from (5.4) by $(P1)$ that $\phi(M_2, M'')$. Then, from $t_1' \lesssim^{\phi}_{M_2} t_2'$ follows by Lemma 5.5.8 that $t_1' \lesssim^{\phi}_{M''} t_2'$. Hence $(\langle s_1' \parallel t_1', M'' \rangle, \langle s_2' \parallel t_2', M'' \rangle) \in \mathcal{R}$.

termination

$s_1 \parallel t_1 \equiv \mathsf{skip}$ only if $s_1 \equiv \mathsf{skip}$ and $t_1 \equiv \mathsf{skip}$. From $s_1 \lesssim^{\phi}_{M} s_2$ follows $\langle s_2, M' \rangle \xrightarrow{\overline{\lambda_1}}{}^{*} \langle \mathsf{skip}, M' \rangle$ where $\widehat{\lambda_1} = \langle\ \rangle$ and from $t_1 \lesssim^{\phi}_{M} t_2$ follows $\langle t_2, M' \rangle \xrightarrow{\overline{\lambda_2}}{}^{*} \langle \mathsf{skip}, M' \rangle$ where $\widehat{\lambda_2} = \langle\ \rangle$.
By (N2) we infer $\langle s_2 \parallel t_2, M' \rangle \xrightarrow{\overline{\lambda_1 \cdot \lambda_2}}{}^{*} \langle \mathsf{skip}, M' \rangle$ where $\widehat{\lambda_1 \cdot \lambda_2} = \langle\ \rangle$ as required. □

Lemma 5.6.5 *Let $\mathbb{S}'$ satisfy $(S1)$ and let ϕ satisfy $(P1), (P2)$ and $(P3)$.*
Let $s_1, s_2 \in \mathbb{S}'$. If $s_1 \lesssim^{\phi}_{M} s_2$ then $!s_1 \lesssim^{\phi}_{M} !s_2$.

Proof
Let $\mathcal{R} = \{(\langle t_1 \parallel !s_1, M\rangle, \langle t_2 \parallel !s_2, M\rangle) \mid t_1 \lesssim^{\phi}_{M} t_2,\ s_1 \lesssim^{\phi}_{M} s_2\} \cup Id_{\mathbb{C}}$.

From transitivity of ϕ (by $(P2)$) and Lemma 5.5.6 follows that $\mathcal{R}$ is interference closed. Furthermore, by Lemma 5.6.4, $\mathcal{R}$ satisfies the following property:

$$\text{If } (\langle s_1, M\rangle, \langle s_2, M\rangle) \in \mathcal{R} \text{ and } t_1 \lesssim^{\phi}_{M} t_2, \text{ then } (\langle t_1 \parallel s_1, M\rangle, \langle t_2 \parallel s_2, M\rangle) \in \mathcal{R} \quad (*)$$

The result follows from Lemma 5.5.14 by showing that $\mathcal{R}$ is a weak ϕ-simulation *up-to* $\lesssim^{\phi}$.
Assume $\phi(M, M')$. We proceed by induction on the depth of the inference.
Consider the following cases
transition
A transition for $\langle t_1 \parallel !s_1, M'\rangle$ can be derived in the following ways:

1. By (N2) from $\langle t_1, M'\rangle \xrightarrow{\lambda} \langle t_1', M''\rangle$, hence, by $(P1)$ and $(P2)$ follows $\phi(M, M'')$. From $t_1 \lesssim^{\phi}_{M} t_2$ follows $\langle t_2, M'\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle t_2', M''\rangle$ such that $t_1' \lesssim^{\phi}_{M''} t_2'$ and $\overline{\lambda'} = \varepsilon^k \cdot \widehat{\lambda}$ for some $k \geq 0$. By (N2) we infer $\langle t_2 \parallel !s_2, M'\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle t_2' \parallel !s_2, M''\rangle$.
   From $(M, M'') \in \phi$ and $s_1 \lesssim^{\phi}_{M} s_2$ we have by Lemma 5.5.8 that $s_1 \lesssim^{\phi}_{M''} s_2$.
   Hence $(\langle t_1' \parallel !s_1, M''\rangle, \langle t_2' \parallel !s_2, M''\rangle \in \lesssim^{\phi} \mathcal{R} \lesssim^{\phi}$.

2. By (N2) from $\langle !s_1, M'\rangle \xrightarrow{\lambda} \langle s_1', M''\rangle$ hence, by $(P1)$ and $(P2)$ follows $\phi(M, M'')$. This transition can be derived in the following ways.

   - By (N6) from $\langle s_1, M'\rangle \xrightarrow{\lambda} \langle s_1', M''\rangle$. From $s_1 \lesssim^{\phi}_{M} s_2$ follows $\langle s_2, M'\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle s_2', M''\rangle$ such that $s_1' \lesssim^{\phi}_{M''} s_2'$ and $\overline{\lambda'} = \varepsilon^k \cdot \widehat{\lambda}$ for some $k \geq 0$. Then, by (N6), we derive $\langle !s_2, M'\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle s_2', M''\rangle$. Using (N2) we infer $\langle t_2 \parallel !s_2, M'\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle t_2 \parallel s_2', M''\rangle$. From $\phi(M, M'')$ and Lemma 5.5.8 follows $t_1 \lesssim^{\phi}_{M''} t_2$. By Lemma 5.6.4 we get $t_1 \parallel s_1' \lesssim^{\phi}_{M''} t_2 \parallel s_2'$. From (E3) and (E8) follows by Lemma 5.6.1 that $t_1 \parallel s_1' \lesssim^{\phi}_{M''} t_1 \parallel s_1 \parallel \mathsf{!skip}$ and $t_2 \parallel s_2' \parallel \mathsf{skip} \lesssim^{\phi}_{M''} t_2 \parallel s_2'$. Hence, $(\langle t_1 \parallel s_1', M''\rangle, \langle t_2 \parallel s_2', M''\rangle) \in \lesssim^{\phi} \mathcal{R} \lesssim^{\phi}$.
   - By (N7) from $\langle s_1 \parallel !s_1, M'\rangle \xrightarrow{\lambda} \langle s_1', M''\rangle$. By the induction hypothesis we get $\langle s_2 \parallel !s_2, M'\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle s_2', M''\rangle$ such that $(\langle s_1', M''\rangle, \langle s_2', M''\rangle) \in \mathcal{R}$ and $\overline{\lambda'} = \varepsilon^k \cdot \widehat{\lambda}$ for some $k \geq 0$. By (N7) we infer $\langle !s_2, M'\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle s_2', M''\rangle$. Using (N2) we derive $\langle t_2 \parallel !s_2, M'\rangle \xrightarrow{\overline{\lambda'}}{}^{*} \langle t_2 \parallel s_2', M''\rangle$. By $\phi(M, M'')$ and Lemma 5.5.8 follows $t_1 \lesssim^{\phi}_{M''} t_2$. Hence, by (*) we get from $(\langle s_1', M''\rangle, \langle s_2', M''\rangle) \in \mathcal{R}$ that $(\langle t_1 \parallel s_1', M''\rangle, \langle t_2 \parallel s_2', M''\rangle) \in \lesssim^{\phi} \mathcal{R} \lesssim^{\phi}$.

3. The proofs of the remaining cases

   - by (N3) from $\langle t_1, M'\rangle \xrightarrow{\lambda} \langle t_1', M''\rangle$ and $\langle !s_1, M'\rangle \xrightarrow{\varepsilon} \langle s_1', M'\rangle$,
   - by (N3) from $\langle t_1, M'\rangle \xrightarrow{\varepsilon} \langle t_1', M'\rangle$ and $\langle !s_1, M'\rangle \xrightarrow{\lambda} \langle s_1', M''\rangle$, and
   - by (N4) from $\langle t_1, M'\rangle \xrightarrow{\sigma_1} \langle t_1', M_1\rangle$ and $\langle !s_1, M'\rangle \xrightarrow{\sigma_2} \langle s', M_2\rangle$ where $M \models \sigma_1 \bowtie \sigma_2$

   are routine combinations of cases 1. and 2.

termination
$t_1 \parallel !s_1 \equiv \mathsf{skip}$ only if $t_1 \equiv \mathsf{skip}$ and $s_1 \equiv \mathsf{skip}$. The proof is analogous to the termination case of Lemma 5.6.4. □

Lemma 5.6.6 *Let $\mathbb{S}'$ satisfy $(S1)$ and let ϕ satisfy $(P1), (P2)$ and $(P3)$.*
Let $s_1, s_2, t_1, t_2 \in \mathbb{S}'$. If $s_1 \lesssim_M^\phi s_2$ and $t_1 \lesssim_M^\phi t_2$, then $c \triangleright s_1[t_1] \lesssim_M^\phi c \triangleright s_2[t_2]$.

Proof By considering the cases $c = \mathit{true}$ and $c = \mathit{false}$ the result follows immediately from Lemma 5.6.1. □

Lemma 5.6.7 *Recursion preserves weak ϕ-refinement.*

Proof Analogous to the proof of Lemma 5.3.15 for strong ϕ-refinement. □

Theorem 5.6.8 *If $\mathbb{S}'$ satisfies $(S1)$ and ϕ satisfies $(P1), (P2)$ and $(P3)$, then $\lesssim_M^\phi$ is a precongruence on $\mathbb{S}'$.*

Proof From Lemmas 5.6.2, 5.6.3, 5.6.4, 5.6.5, 5.6.6 and 5.6.7. □

Corollary 5.6.9 *$\precsim$ is a precongruence on $\mathbb{S}$.*

Proof We need to verify the conditions of Theorem 5.6.8 for $\phi_{stateless} = \mathbb{M} \times \mathbb{M}$. Conditions $(S1), (P1)$ and $(P2)$ are verified in the proof of Corollary 5.3.18. Clearly $Id_{\mathbb{M}} \subset \mathbb{M} \times \mathbb{M}$ which establishes $(P3)$. □

Finally, we address the soundness of weak ϕ-refinement.

Lemma 5.6.10 *Let $\phi \subseteq \mathbb{M} \times \mathbb{M}$ be an interference set.*
If $\langle s, M\rangle \lesssim^{\phi} \langle t, M\rangle$, then $\mathcal{O}^{\phi}(s, M) \subseteq \mathcal{O}^{\phi}(t, M)$.

Proof Analogous to Lemma 5.4.7. □

5.7 Metric Refinement

In this section, we illustrate how the theory from the preceding sections can be used to obtain notions of refinement.

A *metric* is a function that maps program states onto elements of some well-founded set. Metrics are a commonly used for proving termination of programs by the following line of reasoning: If the value of the metric decreases at every execution step and is bounded from below, then we conclude that the program eventually terminates.

As an example of an instantiation of generic refinement, we present a new refinement relation based on metrics. The theory from the previous section asserts that this relation is a precongruence.

Suppose that $T : \mathbb{M} \to \mathbb{N}$ is a metric for a Gamma program P. Metric simulation is obtained as an instance of generic simulation by taking

$$\phi = \{(M, M') \mid T(M') \leq T(M)\} \tag{5.5}$$

We check that this choice for ϕ satisfies the precongruence criteria. Take $\mathbb{S}' = \mathbb{S}_{\mathcal{L}_{(P)}}$; i.e we consider the set of schedules that can be built from (strengthenings of) the rules in P.

(S1) Follows from Lemma 5.3.3.

(P1) Suppose $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$ for some $s \in \mathbb{S}'$, and consider the following cases

- $\lambda = \varepsilon$: Then $M = M'$ hence $T(M) = T(M')$.
- $\lambda = \sigma$: Then by Theorem 3.3.25 follows $\langle P, M\rangle \overset{\sigma}{\rightsquigarrow} \langle P', M'\rangle$. Because T is a metric for P, $T(M') < T(M)$.

(P2) transitivity: if $T(M'') \leq T(M')$ and $T(M') \leq T(M)$, then $T(M'') \leq T(M)$.

(P3) reflexivity: for all M, $T(M) = T(M)$.

Let $\leqslant^T$ and $\lesssim^T$ denote the strong and weak refinement relations obtained by taking ϕ as defined in (5.5). By Theorems 5.3.16 and 5.6.8 follows that $\leqslant^T$ and $\lesssim^T$ are precongruence relations.

Hence any metric that can be used to prove termination of a Gamma program, can be (re-)used to yield a refinement relation that is a precongruence over the set of schedules for that program. For instance, for the summation and prime sieving programs that are presented in Section 7.1 and Section 7.2, the number of elements in the multiset would be a usable metric.

In Chapter 6 we introduce another refinement relation that is obtained as instantiation of generic refinement. This relation induces an additional number of refinement techniques that prove to be very useful in the case studies of Chapter 7.

5.8 Concluding Remarks

We presented a generic framework for the study of simulation-based notions of refinement for our coordination language for Gamma. This theory is based on a definition of simulation which is parameterized by the possible interferences that may be expected from the environment.

This definition of simulation induces a spectrum of possible notions of refinement. This spectrum is partially ordered with respect to subset inclusion of the interference parameter. The statebased and stateless notions of refinement that were developed in Chapter 4 were shown to be instances of the generic notion of refinement that occupy opposites in this spectrum.

Furthermore, we identified conditions on the interference parameter which are sufficient to ensure precongruence of the associated notion of refinement.

6 Convex Refinement

In Chapter 5 we presented a framework which shows that the assumptions about interference from the environment determine the scope of usability of the associated notion of refinement and determine whether the refinement notion is a precongruence or not. We showed that statebased and stateless refinement are opposites in the spectrum of possible assumptions about interference from the environment.

In this chapter, we use the results from Chapter 5 to design a new notion of refinement. This notion makes limited assumptions about interference from the environment and thereby strikes a balance between the assumptions made by statebased and stateless refinement. With this notion of refinement it is possible to justify refinements using properties of the multiset (just as statebased refinement) and use these in a modular way (because it is a precongruence just as stateless refinement). We show the additional refinement laws which are justified by this notion.

6.1 Modelling Interference of a Fixed Context

The behaviour of a schedule depends on the multiset in which it is executed. During execution the context in which a schedule operates may modify the multiset and hence influence the behaviour of that schedule. Different notions of refinement reflect different assumptions about the possible interferences from the context. The theory of generic refinement developed in Chapter 5 allows us to compare notions of refinement with respect to their assumptions about interference.

Next, we describe what kind of assumptions statebased and stateless refinement make about interference from the context. Stateless refinement makes the worst-case assumption: it requires that one schedule can simulate another schedule while any interference (modification of the multiset) is possible (nothing is known about the context). Assuming that the environment may perform an arbitrary (demonic) interference, yields a small refinement relation (only few schedules are related). However, from a practical

point of view, this choice is interesting because the resulting refinement relation is a precongruence.

Statebased refinement makes the opposite assumption: it defines one schedule to be a refinement of another if the latter can match the former's behaviour when no interferences take place. This yields a larger refinement relation (more schedules are related). However, the resulting refinement relation is not a precongruence which make the use of this notion less practical.

In this section we shall develop a notion of refinement based on an intermediate, and often more accurate assumption about the environment. Rather than the "all or nothing" situations we saw for stateless and statebased refinement, we will here assume that the context can only perform a limited set of interferences. The idea behind this assumption is the following.

Consider the situation where P is a simple Gamma program. The derivation of coordination strategies starts with the most general schedule Γ_P. This schedule is refined into more deterministic strategies which use more specialized rewrite rules (strengthenings of the rewrite rules from P). Hence, for these schedules holds that their sort is a strengthening of the sort of P. Now suppose that we want to refine a subschedule s of such a schedule. Then the context in which s operates is also a schedule of P. Hence, the interferences that s may experience from the context are a subset of the rewrites that P can perform.

Hence, this set of interferences can be approximated by all rewrites that P may perform. This assumption suggests the following formalization of the interference parameter in the theory of generic refinement.

Definition 6.1.1 *The interference set of a program P, denoted $\Diamond_P$, is given by the set of pairs (M, M') such that M' can be reached from M by execution of P.*

$$\Diamond_P = \{(M, M') \mid \langle P, M\rangle \overset{\overline{\sigma}}{\rightsquigarrow}{}^* \langle P', M'\rangle\}$$

By taking $\phi = \Diamond_P$, we obtain new notions of refinement where the interferences are limited to the multisets that are reachable by the program P.

Definition 6.1.2

1. *strong convex refinement:* $\leqslant^{\Diamond_P} = \leq^{\phi}$ *where* $\phi = \Diamond_P$
2. *strong convex equivalence:* $=^{\Diamond_P} = =^{\phi}$ *where* $\phi = \Diamond_P$

3. *weak convex refinement:* $\lesssim^{\diamond_P} = \lesssim^{\phi}$ *where* $\phi = \diamondsuit_P$

4. *weak convex equivalence:* $\simeq^{\diamond_P} = \simeq^{\phi}$ *where* $\phi = \diamondsuit_P$

The interferences remain within the execution space of some program. Since this is some closed space, we call this notion *convex* refinement. When it is clear to which program P the refinement notion is associated, we write $\diamondsuit$ instead of $\diamondsuit_P$ (and hence $\leqslant^{\diamond}$ in place of $\leqslant^{\diamond_P}$ and $\lesssim^{\diamond}$ in place of $\lesssim^{\diamond_P}$).

We continue by showing that, for simple Gamma programs, both strong and weak convex refinement satisfy the criteria for precongruence suggested in Section 5.3 and Section 5.6.

Theorem 6.1.3 *Let P be a simple program. Then*

1. $\leqslant^{\diamond_P}$ *is a precongruence over* $\mathbb{S}_{\mathcal{L}(P)}$

2. $\lesssim^{\diamond_P}$ *is a precongruence over* $\mathbb{S}_{\mathcal{L}(P)}$

Proof

1. We check the criteria presented in Section 5.3.

   (S1) By Lemma 5.3.3 follows that $\mathbb{S}_{\mathcal{L}(P)}$ is transition closed.

   (P1) Let $s \in \mathbb{S}_{\mathcal{L}(P)}$. If $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$ then consider the cases

   - $\lambda = \varepsilon$: Then $M = M'$ and by reflexivity of $\rightsquigarrow^*$ follows $\langle P, M\rangle \overset{\langle\rangle}{\rightsquigarrow}{}^* \langle P, M\rangle$.
   - $\lambda = \sigma$: Then, by Lemma 3.3.20, $\langle \Gamma_P, M\rangle \xrightarrow{\sigma} \langle t, M'\rangle$, for some $t \in \mathbb{S}_{\mathcal{L}(P)}$. By Theorem 3.3.25 and Definition 2.1.5 of $\rightsquigarrow^*$ follows $\langle P, M\rangle \overset{\sigma}{\rightsquigarrow}{}^* \langle P, M'\rangle$.

   Hence, by definition of $\diamondsuit_P$, follows $\diamondsuit_P(M, M')$.

   (P2) Transitivity: Suppose $\diamondsuit_P(M, M')$ and $\diamondsuit_P(M', M'')$. By Definition 6.1.1 of $\diamondsuit_P$ follows $\langle P, M\rangle \overset{\bar{\sigma}}{\rightsquigarrow}{}^* \langle P', M'\rangle$ and $\langle P, M'\rangle \overset{\bar{\sigma'}}{\rightsquigarrow}{}^* \langle P'', M''\rangle$. From the semantics in Figure 2.3 follows that, since P is simple, $P' = P'' = P$. Then, by transitivity of $\rightsquigarrow^*$ follows $\langle P, M\rangle \overset{\overline{\sigma\cdot\sigma'}}{\rightsquigarrow}{}^* \langle P, M''\rangle$. Hence, by Definition 6.1.1 of $\diamondsuit_P$, follows $\diamondsuit_P(M, M'')$.

   The result follows from Theorem 5.3.16.

2. In addition to the conditions $(P1)$, $(P2)$ and $(S1)$ that are checked for strong convex refinement (under case 1), we need to check reflexivity (property $(P3)$) of $\diamondsuit_P$ to show precongruence of weak convex refinement.

(P3) Reflexivity: By reflexivity of $\rightsquigarrow^*$ follows for all M, $\langle P, M\rangle \overset{\langle\rangle}{\rightsquigarrow}{}^* \langle P, M\rangle$. Hence, by Definition 6.1.1 of $\Diamond_P$ follows $(M, M) \in \Diamond_P$ for all M.

The result follows from Theorem 5.6.8.

□

Next we show that convex refinement is situated between stateless and statebased refinement. Recall, from Theorems 5.2.11 and 5.2.12, that we have $\phi = \mathbb{M} \times \mathbb{M}$ for stateless refinement and $\phi = Id_{\mathbb{M}}$ for statebased refinement.

Theorem 6.1.4 *For all simple programs P,*

1. $\leqslant \subseteq \leqslant^{\Diamond_P}$ *and* $\leqslant^{\Diamond_P} \subseteq \leqq$
2. $\precsim \subseteq \precsim^{\Diamond_P}$ *and* $\precsim^{\Diamond_P} \subseteq \precapprox$

Proof The set $\Diamond_P = \{(M, M') \mid \langle P, M\rangle \overset{\bar{\sigma}}{\rightsquigarrow}{}^* \langle P, M'\rangle\}$ is used as interference set for convex refinement. From reflexivity of $\rightsquigarrow^*$ follows $Id_{\mathbb{M}} \subseteq \Diamond_P$. Clearly $\Diamond_P \subseteq \mathbb{M} \times \mathbb{M}$. Then, case 1 follows by Theorem 5.2.13 and case 2 follow by Theorem 5.5.16. □

In the next sections we derive some laws for convex refinement. These laws yield new methods for proving refinements on top of the methods we had obtained earlier using statebased and stateless refinement in Chapter 4. Since convex refinement is a precongruence, these laws may be used in a modular way.

6.2 Laws for Convex Refinement

A feature of convex refinement is that it allows properties of the multiset to be used for justifying refinements while taking into account that interferences may occur. In order to use convex refinements, we need methods for reasoning about properties of the multiset. In particular, we need

1. a method for establishing that a property holds at some stage of execution. Such a method deals with the *progress* of a computation.
2. a method for verifying that a property continues to hold if interference occurs. This deals with the *safety* of assuming a property given the possibility of interference.

We postpone reasoning about progress in a setting which allows interference to Section 6.2.3. We proceed by discussing the opportunities that convex refinement provides for reasoning about safety.

A property q "survives" interference from an environment ϕ if $\forall (M, M') \in \phi : [\![q]\!]M \Rightarrow [\![q]\!]M'$. Hence, for the case of convex refinement, a property is preserved by interference if it is preserved by every sequence of transitions of the underlying Gamma program. Formally, a property q is preserved by an environment $\Diamond_P$ if

$$\forall M, M' : [\![q]\!]M \ \wedge \ \langle P, M\rangle \overset{\overline{\sigma}}{\rightsquigarrow}{}^{*} \langle P, M'\rangle \Rightarrow [\![q]\!]M'$$

This is exactly the same as requiring that q is a stable property of the program P.

Hence, by defining the interference set as the reachability set of a program we can verify the preservation of a property by checking that it is a stable property of this program. To this end, we can use the program logic which was presented in Section 2.2.

In the next sections we present a number of convex refinement laws that all use the stability of some property as a premisse. When a multiset appears unbound in a premisses of one of these laws, it should be understood as being universally quantified over the reachable multisets of the program at hand.

6.2.1 Convex Strengthening Laws

The first kind of convex refinements that we look at are concerned with the strengthening of the enabling condition of rewrite rule. Such strengthenings limit the nondeterminism in the selection of elements from the multiset.

In the following lemmas we will use q to represent a property of the program that is used to justify a refinement of the coordination strategy.

Lemma 6.2.1 *Let P be a simple program. Let $s, t \in \mathbb{S}_{\mathcal{L}(P)}$ be schedules and let r and r' be rewrite rules such that $\mathcal{L}(r') \lhd \mathcal{L}(r) \lhd \mathcal{L}(P)$. If*

1. $[\![q]\!]M$
2. *stable q*
3. $\forall M' : \Diamond(M, M') : [\![q]\!]M' \wedge [\![\natural r]\!]M' \Rightarrow [\![\natural r']\!]M'$

then $r' \to s[t] =^{\diamond}_{M} r \to s[t]$.

Proof We show that $r' \to s[t] \leqslant^{\diamond}_{M} r \to s[t]$ and $r \to s[t] \leqslant^{\diamond}_{M} r' \to s[t]$.

- $r' \to s[t] \leqslant^{\diamond}_{M} r \to s[t]$: Assume $\Diamond(M, M')$ and consider the following cases.
  transition

  – If $\langle r' \to s[t], M' \rangle \xrightarrow{\varepsilon} \langle t, M' \rangle$ then $[\![\dagger r']\!]M'$.
  From *stable* q follows $[\![q]\!]M'$. We proceed as follows:

  $$
  \begin{array}{ll}
  & [\![q]\!]M' \wedge [\![\dagger r']\!]M \\
  \Rightarrow & \qquad \neg[\![\natural r']\!]M \Leftrightarrow [\![\dagger r']\!]M, \text{ premisse 3} \\
  & [\![q]\!]M' \wedge \neg([\![q]\!]M' \wedge [\![\natural r]\!]M') \\
  \Leftrightarrow & \qquad \neg[\![\natural r]\!]M \Leftrightarrow [\![\dagger r]\!]M, \text{ De Morgan} \\
  & [\![q]\!]M' \wedge (\neg[\![q]\!]M' \vee [\![\dagger r]\!]M') \\
  \Leftrightarrow & \qquad \wedge \text{ distribution} \\
  & ([\![q]\!]M' \wedge \neg[\![q]\!]M') \vee ([\![q]\!]M' \wedge [\![\dagger r]\!]M') \\
  \Rightarrow & \qquad \text{falsity} \\
  & [\![q]\!]M' \wedge [\![\dagger r]\!]M' \\
  \Rightarrow & \qquad \text{weakening} \\
  & [\![\dagger r]\!]M'
  \end{array}
  $$

  From $[\![\dagger r]\!]M'$ we derive by (N0): $\langle r \to s[t], M' \rangle \xrightarrow{\varepsilon} \langle t, M' \rangle$. By reflexivity of $\leqslant^{\diamond}$ follows $t \leqslant^{\diamond}_{M'} t$.

  – If $\langle r' \to s[t], M' \rangle \xrightarrow{\sigma} \langle s, M'' \rangle$, then $[\![\natural r']\!]M'$. From $r' \lhd r$ follows $[\![\natural r]\!]M'$. Then, by (N1) we derive $\langle r \to s[t], M' \rangle \xrightarrow{\sigma} \langle s, M'' \rangle$. By reflexivity of $\leqslant^{\diamond}$ follows $s \leqslant^{\diamond}_{M'} s$.

  termination: holds vacuously

- $r \to s[t] \leqslant^{\diamond}_{M} r' \to s[t]$: Assume $\Diamond(M, M')$ and consider the following cases.
  transition

  – If $\langle r \to s[t], M' \rangle \xrightarrow{\varepsilon} \langle t, M' \rangle$, then $[\![\dagger r]\!]M'$. From $r' \lhd r$ follows $[\![\dagger r']\!]M'$. Hence by $(N0)$ we derive $\langle r' \to s[t], M' \rangle \xrightarrow{\varepsilon} \langle t, M' \rangle$. By reflexivity of $\leqslant^{\diamond}$ follows $t \leqslant^{\diamond}_{M'} t$.

  – If $\langle r \to s[t], M' \rangle \xrightarrow{\sigma} \langle s, M'' \rangle$, then $[\![\natural r]\!]M'$. From *stable* q follows $[\![q]\!]M'$.
  From $[\![q \wedge \natural r]\!]M'$ follows, by condition 3, that $[\![\natural r']\!]M'$. Then, by $(N1)$ we derive $\langle r' \to s[t], M' \rangle \xrightarrow{\sigma} \langle s, M'' \rangle$. By reflexivity of $\leqslant^{\diamond}$ follows $s \leqslant^{\diamond}_{M''} s$.

  termination: holds vacuously

□

The special case that the property q is invariant leads to the following corollary.

Corollary 6.2.2 *Let P be a simple program and let $s, t \in \mathbb{S}_{\mathcal{L}(P)}$ be schedules. Let r and r' be rewrite rules such that $\mathcal{L}(r') \lhd \mathcal{L}(r) \lhd \mathcal{L}(P)$. If*

1. *invariant q*
2. $\forall M' : \Diamond(M, M') : [\![q]\!]M' \wedge [\![\natural r]\!]M' \Rightarrow [\![\natural r']\!]M'$

then $r' \to s[t] =^{\diamond}_{M} r \to s[t]$.

Proof By Lemma 6.2.1 and the definition of *invariant* . □

We show an example of the application of these laws.

Example 6.2.3 *In Section 7.3 we present a solution to the sorting problem. Input to the sorting program is some sequence $\overline{v} = \langle\, v_1, \ldots, v_N \,\rangle$. This sequence is represented by index-value pairs in the initial multiset: $M_0 = \{(i, v_i) \mid 1 \leq i \leq N\}$. The Gamma program for sorting, as introduced by (2.3), consists of the rewrite rule swap:*

$$swap = (i, x), (j, y) \mapsto (i, y), (j, x) \Leftarrow i < j \wedge x > y \tag{6.1}$$

Let $V = \{v_1, \ldots, v_N\}$ be the multiset of values in the sequence $\overline{v}$. Using the logic from Section 2.2, it is straightforward to show that the multisets of indices and values remain constant throughout execution. Formally,

$$invariant\ \forall i, x : (i, x) : 1 \leq i \leq N \tag{6.2}$$

$$invariant\ \forall i, x : (i, x) : x \in V \tag{6.3}$$

The invariance of these properties ensures that strengthening the enabling condition of the rewrite rule swap with these properties does not affect the outcome of any schedule that swap is part of. To illustrate, we add the predicate "$1 \leq i, j \leq N$" to the rewrite rule swap in the (most general) schedule $S \mathrel{\widehat{=}} !(swap \to S)$.

Define the multiset rewrite rule swap′ by

$$swap' = (i, x), (j, y) \mapsto (i, y), (j, x) \Leftarrow i < j \wedge x > y \wedge 1 \leq i, j \leq N \tag{6.4}$$

The schedule $S' \mathrel{\widehat{=}} !(swap' \to S')$ *is obtained from* S *by replacing the rewrite rule swap by* $swap'$.

Next, we show that $S' =^{\diamond}_{M_0} S$. *To this end, we check the conditions of Corollary 6.2.2. Clearly, swap is a simple program,* $S, S' \in \mathbb{S}_{\mathcal{L}(swap)}$ *and* $swap' \lhd swap$. *The invariance of* $1 \leq i, j \leq N$ *follows from (6.2). Furthermore,* $1 \leq i, j \leq N \wedge \natural swap \Leftrightarrow 1 \leq i, j \leq N \wedge i < j \wedge x > y \Leftrightarrow \natural swap'$.

Corollary 6.2.4 shows another application of Lemma 6.2.1. It shows that if some stable property induces the failure of a rewrite rule, then this rule can be eliminated from the schedule. The proof proceeds in two steps: first, convex refinement is used to show that a rule that can never be executed successfully is equivalent to *fail*; secondly, weak stateless refinement justifies the omission of *fail*.

Corollary 6.2.4 *Let* P *be a simple program and let* $s, t \in \mathbb{S}_{\mathcal{L}(P)}$ *be schedules.*
Let r *be a rule such that* $\mathcal{L}(r) \lhd \mathcal{L}(P)$. *If*

1. $[\![\dagger r]\!]M$
2. *stable* $\dagger r$

then $t \simeq^{\diamond}_{M} r \to s[t]$.

Proof Clearly $fail \lhd r$. Furthermore, from $[\![\dagger r]\!] \wedge [\![\natural r]\!] \Rightarrow false$ (for any M) follows by Lemma 6.2.1 that $fail \to s[t] =^{\diamond}_{M} r \to s[t]$. By Lemma 4.4.35 and Theorem 6.1.4 follows $t \simeq^{\diamond}_{M} fail \to s[t]$. Then by transitivity of $\simeq^{\diamond}_{M}$ follows $t \simeq^{\diamond}_{M} r \to s[t]$. □

Note that if, in Lemma 6.2.4, M is the initial multiset, then $\dagger r$ holds invariantly and $r \to$ may be omitted at any stage of the execution.

The above result can be applied in cases where a property holds which prevents successful execution of a rewrite rule. Program properties can be used in yet another way. If some property ensures successful execution of a rewrite rule, then the next lemma illustrates that the "*else*" branch of rule-conditionals can be omitted. Note that this lemma also provides a method for refining rule-conditional by sequential composition.

Lemma 6.2.5 *Let* P *be a simple program and Let* $s, t \in \mathbb{S}_{\mathcal{L}(P)}$ *be schedules.*
Let r *be a rule such that* $\mathcal{L}(r) \lhd \mathcal{L}(P)$. *If*

1. $[\![\natural r]\!]M$

2. stable $\natural r$

then $r; s =^{\diamond}_{M} r \to s[t]$.

Proof We show $r; s \leqslant^{\diamond}_{M} r \to s[t]$ and $r \to s[t] \leqslant^{\diamond}_{M} r; s$.

- $r; s \leqslant^{\diamond}_{M} r \to s[t]$:
  transition
  Suppose $\Diamond(M, M')$. From *stable* $\natural r$ follows $[\![\natural r]\!]M'$. A transition for the right hand side can only be derived by (N5) from $\langle r, M'\rangle \xrightarrow{\lambda} \langle \mathsf{skip}, M''\rangle$. From $[\![\natural r]\!]M'$ follows that this transition is derived by (N1), hence $\lambda = \sigma$ for some σ. Then for the left hand side we derive, by (N1), $\langle r \to s[t], M'\rangle \xrightarrow{\sigma} \langle s, M''\rangle$.
  By reflexivity of $\leqslant$ follows $s \leqslant^{\diamond}_{M''} s$.
  termination
  This case holds vacuously.

- $r \to s[t] \leqslant^{\diamond}_{M} r; s$:
  transition
  Suppose $\Diamond(M, M')$. From *stable* $\natural r$ follows $[\![\natural r]\!]M'$. Hence, a transition for the left hand side can only be derived by (N1): $\langle r \to s[t], M'\rangle \xrightarrow{\sigma} \langle s, M''\rangle$. Then by (N1) and (N5) we derive $\langle r; s, M'\rangle \xrightarrow{\sigma} \langle s, M''\rangle$ for the right hand side. By reflexivity of $\leqslant$ follows $s \leqslant^{\diamond}_{M''} s$.
  termination
  This case holds vacuously.

□

Note that if, in Lemma 6.2.5, M is the initial multiset, then $\natural r$ holds invariantly, hence the refinement may be applied at any stage of the execution.

In this section we presented refinements that allow for the strengthening of multiset rewrite rules. As a special case, these laws yield a method for the elimination of rewrite rules that can never execute from some stage of the execution onward.

6.2.2 Convex Decomposition Laws

The refinements that we present in this section are called *decomposition laws*. They describe how a (most general) schedule can be decomposed into two (or more) schedules each of which contain strengthenings of the rewrite rules that appear in the original

schedule. Each of these specialized rules takes care of part of the computations of the original rules such that the strengthenings together perform the same computation as the originals.

Such a division is achieved by devising a decomposition of the enabling condition of the original rewrites rule into two (or more) conditions such that the conjunction of these conditions is logically equivalent to the condition of the original rules. Executing these specialized rules in parallel ensures that at their termination, the same properties hold as at termination of the original rules.

Each of the components that results from a decomposition performs a part of the original computation. The robustness against interferences of the computations of these components determines whether they can be executed in parallel or that a sequential ordering is required. If mutual interference between the components is not possible, then the components may be executed in parallel; otherwise, one component must be executed after the other. First, we look at decompositions that allow parallel composition. In the subsequent section we examine decompositions that yield sequentially composed components.

Introducing Parallel Composition

The process of refinement starts from some most general schedule. The refinements of this section replace such a schedule by two or more schedules which have the same form. Hence, these refinements can be used for successive refinements. This gives a repertoire of refinements that can be used for a significant part of the refinement trajectory.

The idea behind the kind of refinement that we examine in this section is the following: if we decompose a schedule $S \mathrel{\widehat{=}} !(r \rightarrow S)$ into multiple components $\Pi_{i=1}^{n} S_i$ where $S_i \mathrel{\widehat{=}} !(r_i \rightarrow S_i)$ and $r_i \lhd r$ for all $i : 1 \leq i \leq n$ such that the r_i's (and hence the components S_i) do not interfere with each other, then executing these components in parallel yields the same result as the single schedule S. The condition of non-interference is formalized by requiring that the property established at termination of one component, may not be invalidated by rewrites of any possible context.

First we show how to decompose a most general schedule into two parallel components. Subsequently, we generalise this result to derive a refinement that enables us to decompose a most general schedule into a number of parallel components.

In the following we will use some notation and properties of most general schedules that were presented in Section 3.3.

In particular, we will use the predicate $\mu_S(s)$ (Definition 3.3.3) to denote that s is a S-derived schedule and use the predicate $[\![\mu_S(s)]\!]M$ to denote that $\langle s, M\rangle$ is a S-derived configuration (Definition 3.3.5).

We show that S-derivedness of a configuration can neither be invalidated by interference from $\Diamond_P$ nor by interference from any schedule $t \in \mathbb{S}_{\mathcal{L}(P)}$ in the context, provided that the disabledness of the constituent rules of S is stable.

Lemma 6.2.6 *Let $P = r_1 + \ldots + r_n$ such that $\forall i : 1 \leq i \leq n :$ stable $\dagger r_i$.*
Let $S \mathrel{\widehat{=}} !(r_1 \to S \parallel \ldots \parallel r_n \to S)$ and let $\langle s, M'\rangle$ be a configuration such that $[\![\mu_S(s)]\!]M'$.

1. *If $\Diamond(M', M'')$, then $[\![\mu_S(s)]\!]M''$,*
2. *If $t \in \mathbb{S}_{\mathcal{L}(P)}$ and $\langle t, M'\rangle \xrightarrow{\overline{\lambda}}{}^* \langle t', M''\rangle$, then $[\![\mu_S(s)]\!]M''$.*

Proof By Definition 3.3.5 follows

(P1) $s \equiv (r_1 \to S)^{a_1} \parallel \ldots \parallel (r_n \to S)^{a_n} \parallel S^k$ with $a_i \geq 0$ and $k \geq 0$

(P2) $(k = 0) \Rightarrow (\forall i : 1 \leq i \leq n : a_i = 0 \;\Rightarrow\; [\![\dagger r_i]\!]M)$

Note that for both case 1 and case 2 we only need to show that $(P2)$ holds for the multiset M' (the form of schedule s does not change).

1. Consider the following cases for k and the a_i's
   - $k \neq 0$ or $\forall i : 1 \leq i \leq n : a_i \neq 0$: Then $(P2)$ holds vacuously.
   - $k = 0$ and $\exists i : 1 \leq i \leq n : a_i = 0$:
     From $[\![\mu_S(s)]\!]M'$ follows $\forall i : 1 \leq i \leq n \wedge a_i = 0 : [\![\dagger r_i]\!]M'$.
     From $\forall i : 1 \leq i \leq n :$ *stable* $\dagger r_i$ follows $\forall i : 1 \leq i \leq n \wedge a_i = 0 : [\![\dagger r_i]\!]M''$.
2. From $\langle t, M'\rangle \xrightarrow{\overline{\lambda}}{}^* \langle t', M''\rangle$ and $t \in \mathbb{S}_{\mathcal{L}(P)}$ follows by Corollary 3.3.26 that $\langle P, M'\rangle \overset{\overline{\lambda''}}{\rightsquigarrow}{}^* \langle P, M''\rangle$ where $\widehat{\lambda''} = \widehat{\lambda'}$. Hence $(M', M'') \in \Diamond_P$. Then, the result follows from part 1 of this lemma.

□

We now prove a refinement which enables the decomposition of a single most general schedule into two separate most general schedules that are composed in parallel.

Lemma 6.2.7 *(Parallel Intro) Let P be a simple program.*
Let $S \mathrel{\widehat{=}} !(r_1 \to S \parallel \ldots \parallel r_n \to S)$ be a schedule such that $\mathcal{L}(S) \lhd \mathcal{L}(P)$.
Let $S_1 \mathrel{\widehat{=}} !(r_{1,1} \to S_1 \parallel \ldots \parallel r_{1,n} \to S_1)$ and let $S_2 \mathrel{\widehat{=}} !(r_{2,1} \to S_2 \parallel \ldots \parallel r_{2,n} \to S_2)$.
If

1. $\forall i : 1 \leq i \leq n : r_{1,i} \lhd r_i$ *and* $r_{2,i} \lhd r_i$
2. $\forall i : 1 \leq i \leq n :$ *stable* $\dagger r_{1,i}$ *and stable* $\dagger r_{2,i}$
3. $\forall i : 1 \leq i \leq n : [\![\natural r_i]\!]M \Rightarrow [\![\natural r_{1,i}]\!]M \vee [\![\natural r_{2,i}]\!]M$

then $S_1 \parallel S_2 \lesssim^{\diamond}_M S$.

Proof Let $\mathcal{R} = \{(\langle s_1 \parallel s_2, M\rangle, \langle s, M\rangle) \mid [\![\mu_{S_1}(s_1)]\!]M, [\![\mu_{S_2}(s_2)]\!]M, \mu^+_S(s)\}$.
We show that $\mathcal{R}$ is a weak convex simulation.
Assume $\Diamond(M, M')$. By Lemma 6.2.6.1 follows $[\![\mu_{S_1}(s_1)]\!]M'$ and $[\![\mu_{S_2}(s_2)]\!]M'$.
Assume $(\langle s_1 \parallel s_2, M\rangle, \langle s, M\rangle) \in \mathcal{R}$ and consider the following cases.
transition
Assume $\langle s_1 \parallel s_2, M'\rangle \xrightarrow{\lambda} \langle t, M''\rangle$. From condition 1 and $\mathcal{L}(S) \lhd \mathcal{L}(P)$ follows $\mathcal{L}(s_1 \parallel s_2) \lhd \mathcal{L}(P)$. Next, consider the following cases for λ:

- $\lambda = \varepsilon$: Then $M'' = M'$ and by reflexivity of $\longrightarrow^*$ follows $\langle s, M'\rangle \xrightarrow{\langle\,\rangle}{}^* \langle s', M'\rangle$ where $s' \equiv s$. Hence $\mu^+_S(s')$.
- $\lambda = \sigma$: From $\forall i : 1 \leq i \leq n : r_{1,i} \lhd r_i$ and $r_{2,i} \lhd r_i$ follows $\mathcal{L}(s_1 \parallel s_2) \lhd \mathcal{L}(S)$. By Theorem 3.3.20 follows $\langle S, M'\rangle \xrightarrow{\sigma} \langle s'', M''\rangle$. From $\mu^+_S(s)$ follows $s \equiv S \parallel s'''$, hence by (N2) we derive $\langle s, M'\rangle \xrightarrow{\sigma} \langle s', M''\rangle$ where $s' \equiv s'' \parallel s'''$. From $\mu_S(s)$, Lemma 3.3.7 and Lemma 3.3.21 follows $\mu^+_S(s')$. From the definition of $\longrightarrow^*$ follows $\langle s, M'\rangle \xrightarrow{\sigma}{}^* \langle s', M''\rangle$.

It remains to show that $t \equiv s'_1 \parallel s'_2$ with $[\![\mu_{S_1}(s'_1)]\!]M''$ and $[\![\mu_{S_2}(s'_2)]\!]M''$. To this end, we consider the possible derivations of $\langle s_1 \parallel s_2, M'\rangle \xrightarrow{\lambda} \langle t, M''\rangle$.

- (N2), (N3): These cases are analogous to the following case (N4).
- By (N4) from $\langle s_1, M'\rangle \xrightarrow{\sigma_1} \langle s'_1, M_1\rangle$ and $\langle s_2, M'\rangle \xrightarrow{\sigma_2} \langle s'_2, M_2\rangle$ where $M' \models \sigma_1 \bowtie \sigma_2$. Hence $t \equiv s'_1 \parallel s'_2$. From Lemma 3.3.7 follows $[\![\mu_{S_1}(s'_1)]\!]M_1$. By Lemma A.2.6 follows that applying σ_1 and σ_2 in arbitrary order yields the same result. Hence σ_2 is enabled in M_1. This implies $\langle s_2, M_1\rangle \xrightarrow{\sigma_2} \langle s'_2, M''\rangle$.

From $\forall i : r_{2,i} \lhd r_i$ and $\mathcal{L}(S) \lhd \mathcal{L}(P)$ follows $\mathcal{L}(s_2) \lhd \mathcal{L}(P)$. Hence by Theorem 3.3.20 follows $\langle \Gamma_P, M_1 \rangle \xrightarrow{\sigma_2} \langle u, M'' \rangle$ for some u. Then by Lemma 3.3.28 follows $\langle P, M_1 \rangle \overset{\sigma_2}{\rightsquigarrow} \langle P, M'' \rangle$. Hence, by definition of $\Diamond_P$ follows $\Diamond_P(M_1, M'')$. Then by Lemma 6.2.6.2 follows $[\![\mu_{S_1}(s_1')]\!]M''$.

Analogously, we derive $[\![\mu_{S_2}(s_2')]\!]M''$.

Hence $(\langle s_1' \parallel s_2', M'' \rangle, \langle s', M'' \rangle) \in \mathcal{R}$.
termination
If $s_1 \parallel s_2 \equiv \mathsf{skip}$ then $s_1 \equiv \mathsf{skip}$ and $s_2 \equiv \mathsf{skip}$. Then by $[\![\mu_{S_1}(s_1)]\!]M$ and $[\![\mu_{S_2}(s_2)]\!]M$ follows $\forall i : 1 \leq i \leq n : [\![\dagger r_{1,i}]\!]M$ and $[\![\dagger r_{2,i}]\!]M$. By $\forall i : 1 \leq i \leq n :$ *stable* $\dagger r_{1,i}$ and *stable* $\dagger r_{2,i}$ follows $\forall i : 1 \leq i \leq n : [\![\dagger r_{1,i} \wedge \dagger r_{2,i}]\!]M'$. By $\forall i : 1 \leq i \leq n : \natural r_i \Rightarrow \natural r_{1,i} \vee \natural r_{2,i}$ follows $\forall i : 1 \leq i \leq n : [\![\dagger r_i]\!]M'$. Then by a straightforward derivation $\langle s, M' \rangle \xrightarrow{\varepsilon}{}^* \langle \mathsf{skip}, M' \rangle$ for any schedule s such that $\mu_S(s)$. □

Lemma 6.2.7 describes how a most general schedule can decomposed into two most general schedules that consist of the same number of rules. The same lemma can be used to decompose a most general schedule into most general schedules that have different numbers of rewrite rules. To this end, the rewrite rules $r_{1,i}$ and $r_{2,i}$ of the resulting schedules should be chosen such that one of them equals *fail*. This rule may subsequently be eliminated from its schedule using Lemma 4.4.35.

Repeatedly decomposing a schedule according to the same strategy yields a uniform (control) structure. In the case of decomposition into parallel components, the resulting structure corresponds to a *forall*-statement. The next lemma enables the introduction of such a structure through a single refinement.

Lemma 6.2.8 *(Parallel Loop Intro) Let P be a simple program.*
Let $S \mathrel{\widehat{=}} !(r_1 \to S \parallel \ldots \parallel r_n \to S)$ be a schedule such that $\mathcal{L}(S) \lhd \mathcal{L}(P)$.
Let $S_i \mathrel{\widehat{=}} !(r_{i,1} \to S_i \parallel \ldots \parallel r_{i,n} \to S_i)$ for all $i : 1 \leq i \leq m$.
Let $L(i) \mathrel{\widehat{=}} i > 0 \rhd (L(i-1) \parallel S_i)$.
If

1. $\forall j : 1 \leq j \leq n : (\forall i : 1 \leq i \leq m : r_{i,j} \lhd r_j)$
2. $\forall j : 1 \leq j \leq n : [\![\natural r_j]\!]M \Rightarrow (\exists i : 1 \leq i \leq m : [\![\natural r_{i,j}]\!]M)$
3. $\forall j : 1 \leq j \leq n : (\forall i : 1 \leq i \leq m :$ *stable* $\dagger r_{i,j})$

then $L(m) \lesssim^{\Diamond}_M S$.

Proof By induction on m.

- $m = 1$: Then from assumption 2 follows $\forall j : 1 \leq j \leq n : [\![\natural r_j]\!]M \Rightarrow [\![\natural r_{1,j}]\!]M$. From premisse 1 follows $r_{1,j} \lhd r_j$, hence $\forall j : 1 \leq j \leq n : (\forall M : [\![\natural r_{1,j}]\!]M \Rightarrow [\![\natural r_j]\!]M)$. Therefore $\forall j : 1 \leq j \leq n : [\![\natural r_j]\!]M \Leftrightarrow [\![\natural r_{1,j}]\!]M$. Then clearly $L(1) \simeq^{\diamond}_{M} S_1 \simeq^{\diamond}_{M} S$.

- $m > 1$: Then $L(m) \simeq L(m-1) \parallel S_m$. Assume that, for all $i : 1 \leq i \leq m$, for all $j : 1 \leq j \leq n$, $r_{i,j} = \overline{x_j} \rightarrow m_j \Leftarrow b_{i,j}$. Let $S' \mathrel{\widehat{=}} !(r'_1 \rightarrow S' \parallel \ldots \parallel r'_n \rightarrow S')$ where $r'_j = \overline{x_j} \rightarrow m_j \Leftarrow b'_j$ with $b'_j \Leftrightarrow (\exists i : 1 \leq i \leq m-1 : b_{i,j})$. Then, by construction,

  1. $\forall j : 1 \leq j \leq n : (\forall i : 1 \leq i \leq m-1 r'_{i,j} \lhd r_i)$
  2. $\forall j : 1 \leq j \leq n : [\![\natural r'_j]\!]M \Leftrightarrow (\exists i : 1 \leq i \leq m-1 : [\![\natural r_{i,j}]\!]M)$

  By assumption 3 follows $\forall j : 1 \leq j \leq n : (\forall i : 1 \leq i \leq m-1 : \ \mathit{stable}\ \dagger r'_{i,j})$.
  Hence by the induction hypothesis follows $L(m-1) \lesssim^{\diamond}_{M} S'$.

  By precongruence of $\lesssim^{\diamond}$ follows $S_m \parallel L(m-1) \lesssim^{\diamond}_{M} S_m \parallel S'$.

  It is straightforward to verify

  1. $\forall j : 1 \leq j \leq n : r'_j \lhd r_j \ \wedge\ r_{m,j} \lhd r_j$.
  2. $\forall j : 1 \leq j \leq n : [\![\natural r_j]\!]M \Rightarrow [\![\natural r'_j]\!]M \vee [\![\natural r_{m,j}]\!]M$
  3. $\forall j : 1 \leq j \leq n : \ \mathit{stable}\ \dagger r'_j \ \wedge\ \mathit{stable}\ \dagger r_{m,j}$.

  Then, from Lemma 6.2.7 follows $S_m \parallel S' \lesssim^{\diamond}_{M} S$.

  By transitivity of $\lesssim^{\diamond}_{M}$ follows $L(m) \lesssim^{\diamond}_{M} S$.

□

In Lemma 6.2.8, schedule L is expressed in a form that emphasizes the analogy with Lemma 6.2.13 which describes a decomposition into sequentially composed components. Alternatively, L may be equivalently expressed as $\Pi_{i=1}^{m} S_i$.

We illustrate the refinement of Lemma 6.2.8 by showing a step in the derivation of a schedule for computing prime numbers. This problem is addressed in full in Section 7.2.

Example 6.2.9 *A Gamma program that computes all primes up to and including N starts with an initial multiset $M_0 = \{2, \ldots, N\}$ and repeatedly eliminates non-primes by executing the rewrite rule sieve:*

$$\mathit{sieve} = c, d \mapsto d \Leftarrow (c \bmod d = 0) \ \wedge\ 2 \leq c \leq N$$

At termination of the program sieve the multiset contains precisely the primes in the interval $[2, N]$.

A data-parallel approach to determining the primes in the interval $[2, N]$ *is to execute* $N - 1$ *tasks in parallel where each of these tasks is responsible for checking primality of one number in the interval* $[2, N]$. *These* $N - 1$ *tasks do not interfere with each other, hence can be executed in parallel.*

Using convex refinement, we can formally derive this approach as follows. The most general schedule for the primes program is $S \mathrel{\widehat{=}} !(sieve \rightarrow S)$. *We define the schedules* S_k *(for all* $k : 2 \leq k \leq N$*) such that* S_k *uses a strengthening* $sieve_k$ *of the rewrite rule sieve to check primality of the integer* k.

$$\begin{array}{lcll} sieve_k & = & c, d \mapsto d \Leftarrow (c \bmod d = 0) \wedge c = k & \text{for } 2 \leq k \leq N \\ S_k & \mathrel{\widehat{=}} & !(sieve_k \rightarrow S_k) & \text{for } 2 \leq k \leq N \end{array}$$

We check the conditions of Lemma 6.2.8:

1. *Clearly,* $\forall k : 2 \leq k \leq N : sieve_k \lhd sieve$

2. $[\![\natural sieve]\!]M \Rightarrow (\exists k : 2 \leq k \leq N : [\![\natural sieve_k]\!]M)$ *follows from invariant* $\forall x : 2 \leq x \leq N$ *because this ensures that variable* c *from sieve is matched to some value in* $[2, N]$.

3. *We have to show that for all* $k : 2 \leq k \leq N :$ *stable* $\dagger sieve_k$. *The termination predicate* $\dagger sieve_k$ *is equivalently expressed as* $\nexists c, d : (c = k) \wedge (c \bmod d = 0)$. *Because the program sieve only removes and never inserts elements, there will never be any elements in the multiset which enable execution of* $sieve_k$.

Hence, $\Pi_{k=2}^{N} S_k \lesssim^{\diamond}_{M_0} S$.

The decomposition laws presented in this section replace a schedule by a parallel composition of (two or more) schedules each of which contains strengthenings of the rewrite rules of the original schedule. This transformation can be thought of as reducing the nondeterminism in the selection of data (fewer combinations of elements from the multiset may be chosen for successful execution of a rewrite rule) in exchange for increasing nondeterminism in the selection of rules (introducing more rewrite rules implies that more choices have to be made about which ones are going to be executed next). In the process of refinement such a trade is desirable because the latter form of nondeterminism can be adequately handled using the stateless methods of refinement, while

nondeterminism in selection of data cannot. Hence, this trade makes it possible to use a simpler method for the elimination of nondeterminism in the selection of data.

A notable characteristic of the semantics of Gamma is that it enforces synchronous termination of the operands of a parallel composition. This termination behaviour is reflected in the most general schedules. Next, we illustrate that Lemma 6.2.8 may be used to break a most general schedule into components that may terminate asynchronously. Loosening synchronization behaviour at termination facilitates rearranging the temporal order of the execution of rewrite rules.

Corollary 6.2.10 *Let P be a simple program.*
Let $S \mathrel{\widehat{=}} !(r_1 \to S \parallel \ldots \parallel r_n \to S)$ be a schedule such that $\mathcal{L}(S) \lhd \mathcal{L}(P)$.
Let $S_i \mathrel{\widehat{=}} !(r_i \to S_i)$ for all $i : 1 \leq i \leq n$ and let $L(i) \mathrel{\widehat{=}} i > 0 \rhd (L(i-1) \parallel S_i)$.
If, for all $i : 1 \leq i \leq n :$ stable $\dagger r_i$, then $L(n) \lesssim^{\diamond}_{M} S$.

Proof Let $S'_i \mathrel{\widehat{=}} !(r_{1,i} \to S'_i \parallel \ldots \parallel r_{n,i} \to S'_i)$ such that $r_{i,j} = r_i$ if $i = j$ and $r_{i,j} = \mathit{fail}$ and $r_{i,j} \lhd r_i$ (i.e. if $r_i = \overline{x_i} \mapsto m_i \Leftarrow b_i$, then $r_{i,j} = \overline{x_i} \mapsto m_i \Leftarrow \mathit{false}$). Now, we check the conditions of Lemma 6.2.8.

1. Let $j : 1 \leq j \leq n$, let $i : 1 \leq i \leq n$. If $i = j$, then $r_{j,i} \lhd r_i$ follows by reflexivity of strengthening. Otherwise, if $i \neq j$, then $r_{i,j} = \mathit{fail}$ and $\mathit{fail} \lhd r_i$ because *fail* is a strengthening of any rewrite rule.

2. Because *false* is unit element for $\vee$, we get $(b_i \vee \mathit{false} \ldots \mathit{false}) \Leftrightarrow b_i$. Since $r_{i,j} = \mathit{fail}$ for all $i, j : i \neq j$, we have $(\exists j : 1 \leq j \leq n : \natural r_{j,i}) \Leftrightarrow \natural r_{i,i}$. The result follows from $r_{i,i} = r_i$.

3. From *stable false* follows *stable* $\dagger r_{i,j}$ for all $i, j : i \neq j$. If $i = j$, then $r_{i,j} = r_i$ and *stable* $\dagger r_i$ follows by assumption.

Hence $\Pi_{i=1}^{n} S'_i \lesssim^{\diamond}_{M} S$.

By Lemma 4.4.35 follows $S_i \sim S'_i$ which eliminates the failing rewrite rules the S_i's. By Theorem 5.2.13 follows $S_i \simeq^{\diamond}_{M} S'_i$. By precongruence and transitivity of $\lesssim^{\diamond}_{M}$ follows $\Pi_{i=1}^{n} S_i \lesssim^{\diamond}_{M} S$. $\square$

Introducing Sequential Composition

In this section we present another refinement for decomposing schedules. This refinement can be used to split a most general schedule into two sequentially ordered components. The first component establishes part of the work that is done by the original schedule. The second component then builds upon the work of the first phase to complete the same task as is performed by the original schedule. This decomposition in two subsequent phases is possible only if interferences leave the result of the first phase unaffected.

The sequential ordering prevents mutual interference among the components that result from decomposition. As a consequence, a weaker condition is required for decomposition into sequential composition than for decomposing into parallel composition (cf. Lemma 6.2.7).

Lemma 6.2.11 *(Sequence Intro) Let P be a simple program.*
Let $S \mathrel{\widehat{=}} !(r_1 \to S \parallel \ldots \parallel r_n \to S)$ and let $S_i \mathrel{\widehat{=}} !(r_{i,1} \to S_i \parallel \ldots \parallel r_{i,n} \to S_i)$ for $i = 1, 2$. If

1. $\forall i : 1 \leq i \leq n : r_{1,i} \lhd r_i$ *and* $r_{2,i} \lhd r_i$
2. $\forall i : 1 \leq i \leq n : [\![\natural r_i]\!]M \Rightarrow [\![\natural r_{1,i}]\!]M \vee \natural[\![r_{2,i}]\!]M$
3. $\forall i : 1 \leq i \leq n :$ *stable* $\dagger r_{1,i}$ *and stable* $\dagger r_{1,i} \wedge \dagger r_{2,i}$

Then $S_1; S_2 \lesssim^{\diamond}_M S$.

Proof Let $TR_1 \Leftrightarrow (\forall j : 1 \leq j \leq n : \dagger r_{1,j})$. Let $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ where

$$\begin{array}{rcll} \mathcal{R}_1 & = & \{(\langle s_1; S_2, M\rangle, \langle s, M\rangle) & \mid \ [\![\mu_{S_1}(s_1)]\!]M,\ s_1 \not\equiv \mathsf{skip},\ \mu^+_S(s)\} \\ \mathcal{R}_2 & = & \{(\langle s_2, M\rangle, \langle s, M\rangle) & \mid \ [\![\mu_{S_2}(s_2)]\!]M,\ [\![TR_1]\!]M,\ \mu^+_S(s)\} \end{array}$$

The proof proceeds by showing that $\mathcal{R}$ is a weak convex simulation up-to weak convex refinement. Suppose $\Diamond(M, M')$. Consider the components of the simulation relation in turn.

$\mathcal{R}_1$: From $\Diamond(M, M')$ and Lemma 6.2.6.1 follows $[\![\mu_{S_1}(s_1)]\!]M'$.
transition: Suppose $\langle s_1, M'\rangle \xrightarrow{\lambda} \langle s_1', M''\rangle$. By Lemma 3.3.7 follows $[\![\mu_{S_1}(s_1')]\!]M''$.
Consider the following cases for λ:

- $\lambda = \varepsilon$: Then $M' = M''$ and $\langle s, M'\rangle \xrightarrow{\langle\,\rangle}{}^* \langle s', M'\rangle$ where $s' \equiv s$.

– $\lambda = \sigma$: From $\forall i : 1 \leq i \leq n : r_{1,i} \lhd r_i$ and Lemma 3.3.17 follows $\mathcal{L}(s_1) \lhd \mathcal{L}(S)$. Then, by Theorem 3.3.20, follows $\langle S, M' \rangle \xrightarrow{\sigma} \langle s'', M'' \rangle$. From $\mu_S^+(s)$ follows $s \equiv s''' \parallel S$. Hence by (N2) follows $\langle s, M' \rangle \xrightarrow{\sigma}{}^* \langle s', M'' \rangle$ where $s' \equiv s'' \parallel s'''$. By Lemma 3.3.7 and Lemma 3.3.21 follows $\mu_S^+(s')$.

Next, to show that the resulting configurations are contained in $\mathcal{R}$, we consider the following cases for s_1'.

– $s_1' \not\equiv \mathsf{skip}$: Then, by $\mathcal{R}_1$, $\langle s_1'; S_2, M'' \rangle \lesssim^\diamond \mathcal{R} \lesssim^\diamond \langle s', M'' \rangle$.

– $s_1' \equiv \mathsf{skip}$: Then $[\![\mu_{S_1}(\mathsf{skip})]\!]M''$ implies $[\![TR_1]\!]M''$. Clearly $[\![\mu_{S_2}(S_2)]\!]M''$ and $\mathsf{skip}; S_2 \lesssim^\diamond S_2$. Hence from $(\langle S_2, M'' \rangle, \langle s', M'' \rangle) \in \mathcal{R}_2 \subseteq \mathcal{R}$ follows $\langle s_1'; S_2, M'' \rangle \lesssim^\diamond \mathcal{R} \lesssim^\diamond \langle s', M'' \rangle$.

termination: Holds vacuously because $s_1 \not\equiv \mathsf{skip}$.

$\mathcal{R}_2$: From $\Diamond(M, M')$ and Lemma 6.2.6.1 follows $[\![\mu_{S_2}(s_2)]\!]M'$. By $\forall i : 1 \leq i \leq n :$ *stable* $\dagger r_{1,i}$ follows *stable* TR_1. Then from $[\![TR_1]\!]M$ follows $[\![TR_1]\!]M'$.
Consider the following cases
transition: Suppose $\langle s_2, M' \rangle \xrightarrow{\lambda} \langle s_2', M'' \rangle$. By Lemma 3.3.7 follows $[\![\mu_{S_2}(s_2')]\!]M''$. By *stable* TR_1 follows $[\![TR_1]\!]M''$. Consider the possible cases for λ:

– $\lambda = \varepsilon$: Then $M'' = M'$ and $\langle s, M' \rangle \xrightarrow{\langle\,\rangle}{}^* \langle s', M' \rangle$ where $s' \equiv s$.

– $\lambda = \sigma$: Analogous to case $\mathcal{R}_1$ follows $\langle s, M' \rangle \xrightarrow{\sigma}{}^* \langle s', M'' \rangle$ where $\mu_S^+(s')$.

From $(\langle s_2', M'' \rangle, \langle s', M'' \rangle) \in \mathcal{R}_2$ follows $\langle s_2', M'' \rangle \lesssim^\diamond \mathcal{R} \lesssim^\diamond \langle s', M'' \rangle$.
termination
Let $(\langle s_1; s_2, M \rangle, \langle s, M \rangle) \in \mathcal{R}$ where $s_1; s_2 \equiv \mathsf{skip}$. Hence $s_1 \equiv \mathsf{skip}$ and $s_2 \equiv \mathsf{skip}$. Because $\mathcal{R}_1$ requires $s_1 \neq \mathsf{skip}$, $(\langle s_1; s_2, M \rangle, \langle s, M \rangle) \in \mathcal{R}_2$. Then, by definition of $\mathcal{R}_2$, follows $[\![TR_1]\!]M$. By *stable* TR_1 follows $[\![TR_1]\!]M'$. From $s_2 \equiv \mathsf{skip}$ and $[\![\mu_{S_2}(s_2)]\!]M$ follows $[\![\forall i : 1 \leq i \leq n : \dagger r_{2,i}]\!]M$. By $\forall i : 1 \leq i \leq n : (\forall M : [\![\natural r_i]\!]M \Rightarrow [\![\natural r_{1,i}]\!]M \vee [\![\natural r_{2,i}]\!]M)$ follows $\forall i : 1 \leq i \leq n : (\forall M : [\![\dagger r_i]\!]M)$. Hence, from $\forall i : 1 \leq i \leq n :$ *stable* $\dagger r_{1,i} \wedge \dagger r_{2,i}$ follows $[\![\forall i : 1 \leq i \leq n : \dagger r_i]\!]M'$. By a straightforward derivation follows, for any s such that $\mu_S^+(s)$, $\langle s, M' \rangle \xrightarrow{\varepsilon}{}^* \langle \mathsf{skip}, M' \rangle$.

□

We use the sorting program (see Section 2.2) to illustrate Lemma 6.2.11.

Example 6.2.12 *Input to the sorting problem is a sequence* $\overline{v} = \langle\, v_1, \ldots, v_N \,\rangle$ *of values. This sequence is represented by the multiset* $M_0 = \{(i, v_i) \mid 1 \leq i \leq N\}$*. The following rewrite rule is proposed for sorting this sequence.*

$$swap = (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow 1 \leq i < j \leq N \wedge x > y$$

The most general schedule for the program is $T \mathrel{\widehat{=}} !(swap \to T)$*. Next, we show how the schedule can be divided into sequential phases based on the following decomposition of the rewrite rule swap.*

We define two schedules T_1 *and* T_2 *that use strengthenings* $swap_1$ *and* $swap_2$ *of swap:*

$$\begin{array}{lcl} swap_1 & = & (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow i = 1 \wedge 1 < j \leq N \wedge x > y \\ T_1 & \mathrel{\widehat{=}} & !(swap_1 \to T_1) \end{array}$$

$$\begin{array}{lcl} swap_2 & = & (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow 2 \leq i < j \leq N \wedge x > y \\ T_2 & \mathrel{\widehat{=}} & !(swap_2 \to T_2) \end{array}$$

Component T_1 *places the smallest value of the interval* $[1, N]$ *at position* 1*. The other component* T_2 *sorts the remaining interval* $[2, N]$*. Hence together they cover the same computation as the original schedule. To justify the convex refinement* $T_1; T_2 \leqslant^{\diamond}_{M_0} T$*, we check the conditions of Lemma 6.2.11:*

1. *clearly* $swap_1, swap_2 \lhd swap$*.*
2. $(1 \leq i < j \leq N) \Rightarrow ((i = 1 \wedge 1 < j \leq N) \vee (2 \leq i < j \leq N))$ *follows straightforwardly by propositional logic*
3. - *Let* $TP_1 = (\forall i, j, x, y : (i,x),(j,y) : (i = 1 \wedge 2 \leq j \leq N) \Rightarrow x \leq y)$*. Then* $\dagger swap_1 \Leftrightarrow TP_1$*. Because no execution of swap can invalidate* TP_1*, follows stable* $\dagger swap_1$*.*
   - *From* $swap_1, swap_2 \lhd swap$ *follows* $\dagger swap \Rightarrow \dagger swap_1 \wedge \dagger swap_2$*. Since obviously stable* $\dagger swap$*, we conclude stable* $\dagger swap_1 \wedge \dagger swap_2$*.*

Note that the correctness of the composition $T_1; T_2$ *depends on the order in which they are executed; the schedule* $T_2; T_1$ *does not guarantee the same result as the original schedule* T*.*

In Example 6.2.12, T_2 solves the same problem we started with, i.e. sorting, but for a problem instance of smaller size. Consequently, T_2 may be refined analogously to

the first refinement, or indeed by any other sorting method we can derive. Repeatedly decomposing according to the same strategy leads to algorithms with a uniform control structure.

Repeating the refinement of Lemma 6.2.11 yields a sequential *for*-loop coordination structure. The following lemma enables the derivation of such a coordination structure in one go. We will see an example of this approach in Section 7.3.4.

Lemma 6.2.13 *(Sequential Loop Intro)*
Let P be a simple program and let $S \mathrel{\widehat{=}} !(r_1 \to S \parallel \ldots \parallel r_n \to S)$.
Let $S_j \mathrel{\widehat{=}} !(r_{j,1} \to S_j \parallel \ldots \parallel r_{j,n} \to S_j)$ for all $j : 1 \leq j \leq m$.
Let $L(i) \mathrel{\widehat{=}} i > 0 \triangleright (L(i-1); S_i)$. If

1. *for all $i : 1 \leq i \leq n : (\forall j : 1 \leq j \leq m : r_{j,i} \lhd r_i)$*
2. *for all $i : 1 \leq i \leq n : [\![\natural r_i]\!]M \Rightarrow (\exists j : 1 \leq j \leq m : [\![\natural r_{j,i}]\!]M)$*
3. *for all $i : 1 \leq i \leq n : (\forall m' : 1 \leq m' \leq m :$ stable $(\forall j : 1 \leq j \leq m' : \dagger r_{j,i}))$*

then $L(m) \lesssim^{\diamond}_M S$.

Proof By induction on m.

- $m = 1$: Then premisse 2 implies $\forall i : 1 \leq i \leq n : [\![\natural r_i]\!]M \Rightarrow [\![\natural r_{1,i}]\!]M$. Furthermore $\mathcal{L}(r_{1,i}) \lhd \mathcal{L}(r_i)$ implies $\forall i : 1 \leq i \leq n : [\![\natural r_{1,i}]\!]M \Rightarrow [\![\natural r_i]\!]M$. Hence $\forall i : 1 \leq i \leq n : [\![\natural r_i]\!]M \Leftrightarrow [\![\natural r_{1,i}]\!]M$ Then clearly, $L(1) \simeq^{\diamond}_M S_1 \simeq^{\diamond}_M S$.

- $m > 1$: Then $L(m) \simeq L(m-1); S_m$. Suppose $r_i = \overline{x_i} \to m_i \Leftarrow b_i$ and $r_{j,i} = \overline{x_i} \to m_i \Leftarrow b_{j,i}$ for all $i : 1 \leq i \leq n$ and $j : 1 \leq j \leq m$. Let $S' \mathrel{\widehat{=}} !(r'_1 \to S' \parallel \ldots \parallel r'_n \to S')$ with $r'_i = \overline{x_i} \to m_i \Leftarrow b'_i$ such that $b'_i \Leftrightarrow (\exists j : 1 \leq j \leq m-1 : b_{j,i})$. Then, by construction

  1. for all $i : 1 \leq i \leq n : (\forall j : 1 \leq j \leq m-1 : r_{j,i} \lhd r'_i)$
  2. for all $i : 1 \leq i \leq n : (\forall M : [\![\natural r'_i]\!]M \Rightarrow (\exists j : 1 \leq j \leq m-1 : [\![\natural r_{j,i}]\!]M))$

  By assumption, premisse 3 holds for $m-1$. Hence, by the induction hypothesis follows $L(m-1) \lesssim^{\diamond}_M S'$.

  By precongruence of $\lesssim^{\diamond}_M$ then follows $L(m-1); S_m \lesssim^{\diamond}_M S'; S_m$.

  It is straightforward to verify

1. for all $i : 1 \leq i \leq n : r'_i \lhd r_i$ and $r_{m,i} \lhd r_i$
2. for all $i : 1 \leq i \leq n : [\![\natural r_i]\!]M \Rightarrow [\![\natural r'_i]\!]M \vee [\![\natural r_{m,i}]\!]M$
3. for all $i : 1 \leq i \leq n :$ *stable* $\dagger r'_i$ and *stable* $\dagger r'_i \wedge \dagger r_{m,i}$

Hence, by Lemma 6.2.11 follows $S'; S_m \lesssim^{\diamond}_M S$.

By transitivity of $\lesssim^{\diamond}_M$ follows $L(m) \lesssim^{\diamond}_M S$.

□

Straightforward variations of Lemma 6.2.13 can be obtained by varying the numbering of the components; e.g. by taking $L'(i) \mathrel{\widehat{=}} i < m \rhd S_i \; ; L'(i+1)$, one can prove $L'(1) \lesssim^{\diamond}_M S$.

6.2.3 Progress

The convex refinement laws of Section 6.2 depend on properties of the multiset. In some cases, the required properties hold invariantly. More often, these properties do not yet hold at the beginning of execution but are established by execution of one or more preceding schedules.

For example, suppose we want use $s' \leqslant^{\diamond}_M s$ to refine s in $t; s$ and the refinement depends on $[\![q]\!]M$. Then we need a method for establishing whether execution of t modifies the multiset such that q holds; i.e. we need to be able to reason about the outcomes of schedule t.

To this end, we instantiate the output function (from Definition 5.4.5) for convex refinement.

Definition 6.2.14 *The output function on configurations under convex interference, denoted $\mathcal{O}^{\diamond_P}$, is defined by $\mathcal{O}^{\phi}$ where $\phi = \Diamond_P = \{(M, M') \mid \langle P, M\rangle \overset{\overline{\lambda}}{\rightsquigarrow}{}^* \langle P, M'\rangle\}$ (for some simple program P). We write $\mathcal{O}^{\diamond}$ if the program P is clear from the context.*

The next lemma provides a method for refining parts of a schedule that appear to the right of a sequential composition (it is a generalization of Lemma 4.3.12 to a setting where "convex" interference is possible).

Lemma 6.2.15 *Let P be a simple program. Let $s_1, s_2, t \in \mathbb{S}_{\mathcal{L}(P)}$.*
If $\forall M' : M' \in \mathcal{O}^{\diamond}(t, M) : s_1 \leqslant^{\diamond}_{M'} s_2$, then $t; s_1 \leqslant^{\diamond}_M t; s_2$.

Proof Let $\mathcal{R} = \{(\langle t; s_1, M\rangle, \langle t; s_2, M\rangle) \mid \forall M' \in \mathcal{O}^\diamond(t, M) : s_1 \leqslant^\diamond_{M'} s_2 \}$.
We show that $\mathcal{R}$ is a strong convex simulation.
Suppose $\langle t; s_1, M\rangle \mathcal{R} \langle t; s_2, M\rangle$ and $\Diamond(M, M')$. Since $\Diamond$ is reflexive and transitive, we get by Lemma 5.4.6 that $\mathcal{O}^\diamond(t, M') \subseteq \mathcal{O}^\diamond(t, M)$. Consider the following cases
transition
A transition can be derived in the following ways

- by (N5), if $t \not\equiv \mathsf{skip}$, from $\langle t, M'\rangle \xrightarrow{\lambda} \langle t', M''\rangle$. Then by (N5) $\langle t; s, M'\rangle \xrightarrow{\lambda} \langle t'; s, M''\rangle$. By Lemma 5.4.6 follows $\mathcal{O}^\diamond(t', M'') \subseteq \mathcal{O}^\diamond(t, M')$. By transitivity of $\subseteq$ follows $\forall M' \in \mathcal{O}^\diamond(t', M'') : s_1 \leqslant^\diamond_{M'} s_2$. Hence $\langle t'; s', M''\rangle \mathcal{R} \langle t'; s, M''\rangle$.

- by (N9), if $t \equiv \mathsf{skip}$ and $s_1 \not\equiv \mathsf{skip}$, from $\langle s_1, M'\rangle \xrightarrow{\lambda} \langle s_1', M''\rangle$. From $t \equiv \mathsf{skip}$ follows $M' \in \mathcal{O}^\diamond(t, M)$, hence $s_1 \leqslant^\diamond_{M'} s_2$. Then $\langle s_2, M'\rangle \xrightarrow{\lambda} \langle s_2', M''\rangle$ such that $s_1' \leqslant^\diamond_{M''} s_2'$. From $t \equiv \mathsf{skip}$ and the definition of $\mathcal{O}^\diamond$, follows $N \in \mathcal{O}^\diamond(t, M'') \Leftrightarrow \Diamond(M'', N)$. By transition-closedness of $\leqslant^\diamond$ follows that $\forall N : s_1' \leqslant^\diamond_{M''} s_2' \wedge \Diamond(M'', N) \Rightarrow s_1' \leqslant^\diamond_N s_2'$. Hence $\forall N \in \mathcal{O}^\diamond(t, M'') : s_1' \leqslant^\diamond_N s_2'$. Then $\langle t; s_1', M''\rangle \mathcal{R} \langle t; s_2', M''\rangle$.

termination
$t; s' \equiv \mathsf{skip}$ implies $t \equiv \mathsf{skip}$ and $s' \equiv \mathsf{skip}$. From $t \equiv \mathsf{skip}$ follows that $s' \leqslant^\diamond_M s$. Then, from $s' \equiv \mathsf{skip}$ follows $s \equiv \mathsf{skip}$, hence $t; s \equiv \mathsf{skip}$. □

A method of using Lemma 6.2.15, that will be illustrated in Chapter 7, consists of establishing an intermediate property, say q, that captures those properties of the set of outputs of schedule t that are required for justifying the refinement $s_1 \leqslant s_2$. More precisely, this method proceeds through the following steps

1. Show that the multisets that are output of schedule t satisfy some property, say q.
2. Show that q is not invalidated by any interference that may occur after termination of t and before execution of s_1 (or s_2).
3. Show that if a multiset M satiesfies q, then $s_1 \leqslant^\diamond_M s_2$.

This line of reasoning is formalised by Corollary 6.2.16.

Corollary 6.2.16 *Let P be a simple program. Let $s_1, s_2, t \in \mathbb{S}_{\mathcal{L}(P)}$.*
If

1. $\forall M' : M' \in \mathcal{O}^\diamond(t, M) : [\![q]\!] M'$

2. *stable* q

3. $\forall M : [\![q]\!]M \Rightarrow s_1 \leqslant^{\diamond}_M s_2$

then $t; s_1 \leqslant^{\diamond}_M t; s_2$.

Proof By Lemma 6.2.15. □

For establishing the first premisse of Corollary 6.2.16, we are faced with the task of determining whether, given some initial multiset M, condition q holds after execution of t under all possible interferences by some simple program. Although is possible to establish such properties using simulations, this is not an ideal technique because it incites operational reasoning which is relatively error-prone. As alternative methods for reasoning about progress of schedules, we may resort to Lemma 3.3.31. This lemma shows that the properties which hold at termination of most general schedules can be derived in a syntactical manner.

However, Lemma 3.3.31 does not take interference into account. Example 6.2.17 illustrates that due to this omission, this method does not carry over to situations where interference is possible (as is the case for convex refinement).

Example 6.2.17 *Let P be a simple program. Let $S \mathrel{\widehat{=}} !(r_1 \to S \,\|\, r_2 \to S)$ such that $\mathcal{L}(S) \lhd \mathcal{L}(P)$. Execution of S starts in some multiset M. Because S is a most general schedule, the following properties hold for any configuration $\langle s, M' \rangle$ that $\langle S, M \rangle$ evolves into*

1. *s is of the form $(r_1 \to S)^{a_1} \,\|\, (r_2 \to S)^{a_2} \,\|\, S^k$*

2. *if $k = 0$, then $a_i = 0$ implies $[\![\dagger r_i]\!]M'$ for $i : 1 \leq i \leq 2$*

Assume that the system arrives at a schedule $s \equiv r_1 \to S$ (hence $a_1 = 1$, $a_2 = 0$ and $k = 0$). Then, from property 2 follows $[\![\dagger r_2]\!]M'$. Now, suppose that an interference $\Diamond_P(M', M'')$ takes place which changes the multiset M' into M'' such that $[\![\natural r_2]\!]M''$ and $[\![\dagger r_1]\!]M''$. Then, by (N0), the following transition can be made $\langle r_1 \to S, M'' \rangle \xrightarrow{\varepsilon} \langle \mathsf{skip}, M'' \rangle$. Thus, the most general schedule S terminates in a multiset M'' which does not satisfy $\dagger r_1 \wedge \dagger r_2$.

This example shows that the relation between the form of the schedule and the disabledness of its rules no longer holds if the multiset is susceptible to arbitrary interference. In particular this relation does not hold at termination while this is assumed by the method of Lemma 3.3.31.

The harm seems to come from the fact that the interference may invalidate the relation between the form of the schedule and the disabledness of its rules. Lemma 6.2.18 shows that this relation can be retained by requiring that the disabledness of the rules may not be invalidated by interference.

Lemma 6.2.18 *Let P be a simple program. Let $S \mathrel{\widehat{=}} !(r_1 \to S \parallel \ldots \parallel r_n \to S)$ for $n \geq 1$ such that $\mathcal{L}(S) \lhd \mathcal{L}(P)$. If $\forall i : 1 \leq i \leq n :$ stable $\dagger r_i$, then $\forall M' : M' \in \mathcal{O}^\diamond(S, M) : [\![(\forall i : 1 \leq i \leq n : \dagger r_i)]\!] M'$.*

Proof If $M' \in \mathcal{O}^\diamond(S, M)$, then there is a sequence of transitions and interferences from $\langle S, M\rangle$ to $\langle \mathsf{skip}, M'\rangle$. Hence, in this sequence, every rewrite rule r_i must have executed and failed at least once. Hence, for all i, there is a multiset in this sequence such that $\dagger r_i$. By *stable* $\dagger r_i$ follows that $\dagger r_i$ continues to hold from this stage of execution onward. Hence for the multiset M' from the final configuration holds $\forall i : 1 \leq i \leq n : [\![\dagger r_i]\!] M'$. □

Using Lemma 6.2.18 we obtain Theorem 6.2.19 as a special case of Lemma 6.2.15.

Theorem 6.2.19 *Let P be a simple program. Let $S \mathrel{\widehat{=}} !(r_1 \to S \parallel \ldots \parallel r_n \to S)$ for $n \geq 1$ where $\mathcal{L}(S) \lhd \mathcal{L}(P)$. If*

1. *$\forall i : 1 \leq i \leq n :$ stable $\dagger r_i$*
2. *$[\![\forall i : 1 \leq i \leq n : \dagger r_i]\!] M \Rightarrow s_1 \leqslant^\diamond_M s_2$*

then $S; s_1 \leqslant^\diamond_M S; s_2$

Proof From Lemma 6.2.15 and Lemma 6.2.18. □

The conclusion of Theorem 6.2.19 can be generalized to $S; u; s' \leqslant^\diamond_M S; u; s$ provided q is *stable* under u. This is necessarily so if u is a proper schedule of P (i.e. $\mathcal{L}(u) \lhd \mathcal{L}(P)$).

We have seen how properties that are established by a schedule on the left hand side of a sequential composition may be used for justifying refinements of a schedule on the right hand side of that composition. Besides sequential composition, the rule-conditional "$r \to ..[..]$" construct also imposes a strict precedence ordering on the execution of rewrite rules. Next, we present a lemma that enables us to prove refinements using properties that can be derived from the execution of a single rewrite rule.

Lemma 6.2.20 *Let P be a simple program. Let r be a rule such that $\mathcal{L}(r) \lhd \mathcal{L}(P)$. Let $s, t \in \mathbb{S}_{\mathcal{L}(P)}$. If $\forall M' : M' \in \mathcal{O}^\diamond(r, M) : s' \leqslant^\diamond_M s$, then*

1. $r \to s'[t] \leqslant^{\diamond}_{M} r \to s[t]$
2. $r \to t[s'] \leqslant^{\diamond}_{M} r \to t[s]$

Proof

1. Let $\mathcal{R} = \{(\langle r \to s'[t], M\rangle, \langle r \to s[t], M\rangle) \mid \forall M' \in \mathcal{O}^{\diamond}(r, M) : s' \leqslant^{\diamond}_{M'} s)\ \}$
   $\cup\ \{(\langle s', M\rangle, \langle s, M\rangle) \mid s' \leqslant^{\diamond}_{M} s\}$

   We show that $\mathcal{R}$ is a strong convex simulation.
   By definition of $\leqslant^{\diamond}_{M}$ follows that the second component is a strong convex simulation. We consider the remaining component.
   Assume $\Diamond(M, M')$. By Lemma 5.4.6 follows that $\mathcal{O}^{\diamond}(r, M') \subseteq \mathcal{O}^{\diamond}(r, M)$.
   transition
   Suppose $\langle r \to s'[t], M'\rangle \xrightarrow{\lambda} \langle u, M''\rangle$, hence $M'' \in \mathcal{O}^{\diamond}(r, M')$. By transitivity of $\subseteq$ follows $M'' \in \mathcal{O}^{\diamond}(r, M)$. This transition can be derived by

   - (N0), then $\lambda = \varepsilon$, $M'' = M'$ and $u = t$. Then $\langle r \to s[t], M'\rangle \xrightarrow{\varepsilon} \langle t, M'\rangle$.
     By reflexivity of $\leqslant^{\diamond}$ follows $t \leqslant^{\diamond}_{M'} t$, hence $(\langle t, M'\rangle, \langle t, M'\rangle) \in \mathcal{R}$.
   - (N1), then $\lambda = \sigma$ and $u = s'$. Then, by (N1), $\langle r \to s[t], M'\rangle \xrightarrow{\sigma} \langle s, M''\rangle$.
     From $M'' \in \mathcal{O}^{\diamond}(r, M)$ follows $s' \leqslant^{\diamond}_{M''} s$, hence $(\langle s', M''\rangle, \langle s, M''\rangle) \in \mathcal{R}$.

   termination Holds vacuously.

2. Analogous to case 1.

□

Analogous to Corollary 6.2.16, Lemma 6.2.20 may be applied using an intermediate property q. This is described by Corollary 6.2.21.

Corollary 6.2.21 *Let P be a simple program. Let r be a rule such that $\mathcal{L}(r) \lhd \mathcal{L}(P)$. Let $s, t \in \mathbb{S}_{\mathcal{L}(P)}$. If*

1. $\forall M' : M' \in \mathcal{O}^{\diamond}(r, M) : [\![q]\!]M'$
2. *stable q*
3. $\forall M : [\![q]\!]M \Rightarrow s' \leqslant^{\diamond}_{M} s$

then

1. $r \to s'[t] \leqslant^{\diamond}_{M} r \to s[t]$
2. $r \to t[s'] \leqslant^{\diamond}_{M} r \to t[s]$

Proof By Lemma 6.2.20. □

Lemma 6.2.22 generalises Lemma 6.2.20 to deal with equivalent schedules.

Lemma 6.2.22 *Let P be a simple program. Let r be a rule such that $\mathcal{L}(r) \lhd \mathcal{L}(P)$. Let $s, t \in \mathbb{S}_{\mathcal{L}(P)}$. If $\forall M' : M' \in \mathcal{O}^{\diamond}(r, M) : s' =^{\diamond}_{M} s$, then*

1. $r \to s'[t] =^{\diamond}_{M} r \to s[t]$
2. $r \to t[s'] =^{\diamond}_{M} r \to t[s]$

Proof From Lemma 6.2.20 and $=^{\diamond}_{M} \; = \; \leqslant^{\diamond}_{M} \cap \leqslant^{\diamond}_{M}{}^{-1}$. □

These results enable us to prove refinements which deal with rewrite rules that disable their own execution (as promised at the end of Section 6.2). First, we consider the case that a rule is invariably disabled. Then, we prove the case that a rule disables its own subsequent successful execution.

Lemma 6.2.23 *Let P be a simple program. Let $S \mathrel{\widehat{=}} !(r \to S)$ where $\mathcal{L}(r) \lhd \mathcal{L}(P)$. If*

1. $[\![\dagger r]\!] M$
2. *stable* $\dagger r$

then $\mathsf{skip} \simeq^{\diamond}_{M} S$.

Proof

$$
\begin{array}{ll}
 & \mathsf{skip} \\
\simeq^{\diamond}_{M} & \qquad (E8) \text{ and Lemma 5.3.7} \\
 & !\mathsf{skip} \\
\simeq^{\diamond}_{M} & \qquad \text{Corollary 6.2.4} \\
 & !r \to S \\
\simeq^{\diamond}_{M} & \qquad \text{Lemma 5.6.1, def. } S \\
 & S
\end{array}
$$

□

Lemma 6.2.24 *Let P be a simple program. Let $S \mathrel{\widehat{=}} !(r \to S)$ where $\mathcal{L}(r) \lhd \mathcal{L}(P)$. If*

1. $\forall M' : M' \in \mathcal{O}^\diamond(r, M) : [\![\dagger r]\!] M'$
2. *stable* $\dagger r$

then $r \lesssim^\diamond_M S$.

Proof

$$
\begin{array}{lll}
 & r \to \mathsf{skip} & \\
\simeq^\diamond_M & & \text{Lemmas 6.2.22 and 6.2.23} \\
 & r \to S & \\
\lesssim^\diamond_M & & s \lesssim^\diamond !s \\
 & !(r \to S) & \\
\simeq^\diamond_M & & \text{Lemma 5.6.1, def. } S \\
 & S &
\end{array}
$$

□

6.3 Concluding Remarks

Based on insights from the generic theory from Chapter 5, we developed in this chapter a new precongruent notion of refinement, called *convex refinement*. The basic idea behind this notion is to approximate the interference that a schedule may experience by the rewrites that the program for which the schedule is designed may make. This ensures that all safety properties that a program satisfies, also hold throughout execution of a schedule (for this program). Hence, these program properties can be used for proving convex refinements of schedules. Since it is considerably easier to establish properties of programs compared to properties of schedules, this reduces the complexity of reasoning about refinement.

We have derived a number of new refinement laws which allow the specialization of rewrite rules based on properties of the multiset and laws which enable the introduction of parallel and sequential loop structures. The collection of convex laws is not aimed to be complete. The laws can be extended as the need arises.

Convex refinement has in common with statebased refinement that it allows properties of the multiset to be used for proving refinements. Additionally, it shares with stateless refinement that it is a precongruence. Consequently, its refinements may be

used in a modular way. Hence, convex refinement combines the useful features from the other notions of refinement. Illustrations of the use of convex refinement are presented in Chapter 7.

Convex refinement is the last variant of a refinement relation that we develop in this thesis. We next discuss how the notions of refinement that we have presented relate to each other. To start, the following table gives an overview of the notions of refinement and how they can be instantiated from generic refinement.

generic simulation	$(M, M') \in \Phi$ iff	precongruence	history preserving
stateless simulation	true	yes	no
metric simulation	$T(M') \leq T(M)$	yes	no
convex simulation	$\langle P, M\rangle \overset{\overline{\sigma}}{\rightsquigarrow}{}^{*} \langle P, M'\rangle$	yes	yes
statebased simulation	$M' = M$	no	yes

Theorem 5.2.13 shows that these refinement relations are order by subset inclusion of the interference parameter. Additionally, Theorem 5.5.18 proves that the strong notions of refinement are contained in the corresponding weak notions. Combining these results yields an ordering on the notions of refinement that is depicted by Figure 6.1.

Figure 6.1: Lattice of Notions of Refinement

Next, we discuss why these inclusions are strict.

The fact the inclusion of strong refinement in weak refinement is strict follows from Lemma 4.4.35. This lemma provides an example of a weak stateless refinement which is not a strong stateless refinement.

The containment of stateless refinement within convex refinement and convex refinement within statebased refinement follows from Theorem 5.2.13. The fact that these inclusions are strict follows from the following refinement. $\langle \mathit{fail}, \{\}\rangle \leqslant^{\diamond} \langle \mathit{add}, \{\}\rangle$ is a strong convex refinement, but not a strong stateless refinement.

Finally, we consider convex and statebased refinement.

Consider the sorting program *swap* and a multiset $M = \{(1, C), (2, A), (3, B)\}$ which represents the sequence $\langle\, C, A, B\,\rangle$. Let $\mathit{swap}_{k,l}$ be the a strengthening of *swap* defined by

$$\mathit{swap}_{k,l} = (i, x), (j, y) \mapsto (i, y), (j, x) \Leftarrow i < j \wedge i = k \wedge j = l \wedge x > y$$

Then $\langle \mathit{swap}_{1,2}; \mathit{swap}_{2,3}, M\rangle \leqq \langle \mathit{swap}_{1,2} \rightarrow \mathit{swap}_{2,3}, M\rangle$ is a strong statebased refinement (and even an equivalence), but not a strong convex refinement.

7 Case Studies

In this chapter we will illustrate the method of program design proposed in the previous chapters by considering a number of case studies.

The problems we study cover diverse applications in computing science: Summation is an elementary problem that is used as an introductory case study. Sorting and computing primes are well-known problems that are widely used for illustrating formal methods of program development. As more advanced cases we discuss a combinatorial problem: computing the shortest paths to some node in a graph, and a problem from scientific computing: solving triangular systems of linear equations.

For each case study we proceed through the following series of steps.

1. We start with a brief description of the problem under study.

2. Next, we present a Gamma program for the problem at hand and address its correctness. The correctness of a Gamma program may either follow by construction if the method from [12] is followed. Alternatively, the programming logic described in Chapter 2 may be used for a-posteri verification of the program's correctness.

3. Subsequently, we construct the most general schedule for the Gamma program. This schedule serves as an initial specification of the coordination strategy for the Gamma program.

4. One or more avenues for deriving coordination strategies by successive stepwise refinement from the most general schedule are investigated.

5. At the end of each case study, the relationship between the coordination strategies that have been derived is discussed as well as the strengths and weaknesses of the methods used for their derivation.

7.1 Summation

As a gentle introduction to the method of derivation, we start with a straightforward example: summation. The Gamma program for summation consists of the following rewrite rule

$$add \widehat{=} x, y \mapsto x + y$$

A correctness proof of this program can be found in [24].

7.1.1 Coordination Strategies for Summation

The most general schedule for the program *add* is

$$\Gamma_{add} \widehat{=} !(add \rightarrow \Gamma_{add})$$

Suppose that the initial multiset in which the *add* program is started, contains n numbers. We can use this information to adapt the schedule for summation such that it performs exactly the necessary $n-1$ additions. At this stage, we will not yet impose any (sequential) order on the computation. The schedule we consider here is simply add^{n-1}. Hence we are interested in the following refinement

Lemma 7.1.1 $\langle add^{n-1}, M\rangle \precsim\approx \langle \Gamma_{add}, M\rangle$ *with* $\#M = n$

Proof Let $\mathcal{R} = \{(\langle add^{i-1}, M\rangle, \langle \Gamma^k_{add}, M\rangle) \mid \#M = i, i \geq 0, k \geq 1\}$.
We prove that $\mathcal{R}$ is a weak statebased simulation.
transition
Suppose $\langle add^{i-1}, M\rangle \xrightarrow{\lambda} \langle add^{i'}, M'\rangle$. Then $add^{i-1} \not\equiv \mathsf{skip}$, hence $i > 1$.
Consider the following cases.

- $\lambda = \varepsilon$: An ε-transition is derived by (N0) from $\langle add, M\rangle\surd$. This implies that $i \leq 1$ which contradicts $i > 1$.
- $\lambda = \sigma$: Then by Lemma 3.2.5 there exists a sequence

  $$\langle add^{i_0}, M_0\rangle \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_m} \langle add^{i_m}, M_m\rangle$$

  of single-step transitions such that $\langle add^{i_0}, M_0\rangle = \langle add^{i-1}, M\rangle$ and $\langle add^{i_m}, M_m\rangle = \langle add^{i'}, M'\rangle$. A single-step transition $\langle add^{n_j}, M_j\rangle \xrightarrow{\sigma_j} \langle add^{n_{j+1}}, M_{j+1}\rangle$, is derived using (N2) and (N1) from $\langle add, M_j\rangle \xrightarrow{\sigma_j} \langle \mathsf{skip}, M_j[\sigma_j]\rangle$ where $\sigma_j = \{x + y\}/\{x, y\}$.

Hence $n_{j+1} = n_j - 1$ and $\#M_{j+1} = \#M_j - 1$. By definition of $\mathcal{R}$ holds $\#M_0 = i_0 + 1$. Hence by induction on the length of the transition sequence follows that $\#M_m = i_m + 1$. Hence, for $\langle add^{i'}, M' \rangle$ holds that $\#M' = i' + 1$.

Furthermore, by Lemma 3.3.20 and Lemma 3.3.22 follows $\langle \Gamma_{add}, M \rangle \xrightarrow{\sigma} \langle \Gamma^{k'}_{add}, M' \rangle$ for some $k' \geq 1$. Then by (N2) and definition of $\longrightarrow^*$: $\langle \Gamma^{k}_{add}, M \rangle \xrightarrow{\sigma}{}^* \langle \Gamma^{k-1+k'}_{add}, M' \rangle$ where $k - 1 + k' \geq 1$. Then $(\langle add^{i'}, M' \rangle, \langle \Gamma^{k}_{add}, M' \rangle) \in \mathcal{R}$.

termination
$add^{i-1} \equiv \mathsf{skip}$ if $i \leq 1$. Then, by the definition of $\mathcal{R}$ follows $\#M \leq 1$, hence $\langle add, M \rangle\surd$. Then by $(N0)$, $(N6)$ and $(N8)$ we derive $\langle \Gamma_{add}, M \rangle \xrightarrow{\varepsilon} \langle \mathsf{skip}, M \rangle$. By definition of $\longrightarrow^*$ follows $\langle \Gamma_{add}, M \rangle \xrightarrow{\varepsilon}{}^* \langle \mathsf{skip}, M \rangle$. Clearly $\widehat{\varepsilon} = \langle\ \rangle$. □

We can further refine the summation strategy into one that imposes recursive doubling-style behaviour. A schedule which describes such behaviour can be defined as follows.

$$RecDubSum(i) \mathrel{\widehat{=}} (i > 1) \triangleright (add^{\lfloor i/2 \rfloor}; RecDubSum(\lceil i/2 \rceil))$$

Lemma 7.1.2 uses algebraic reasoning to show that the recursive doubling strategy refines the unordered summation strategy.

Lemma 7.1.2 *For all n, $RecDubSum(n) \leqslant add^{n-1}$*

Proof By induction on n.

- $n \leq 1$: $RecDubSum(n) \simeq \mathsf{skip} \simeq add^{n-1}$ as required.

- $n > 1$:

$$\begin{array}{lll} & RecDubSum(n) & \\ \simeq & & \text{def. } RecDubSum \\ & add^{\lfloor n/2 \rfloor}; RecDubSum(\lceil n/2 \rceil) & \\ \leqslant & & \text{Induction Hypothesis} \\ & add^{\lfloor n/2 \rfloor}; add^{\lceil n/2 \rceil - 1} & \\ \leqslant & & \text{Corollary 4.4.14.1} \\ & add^{n-1} & \end{array}$$

□

Subsequently, we may refine $RecDubSum(n)$ into a sequential schedule. A sequential summation strategy can be defined as follows.

$$Sum(n) \mathrel{\widehat{=}} n > 1 \triangleright (add; Sum(n-1))$$

Next, we prove that the sequential schedule $Sum(n)$ is a refinement of the recursive doubling schedule $RecDubSum(n)$.

Lemma 7.1.3 *For all n, $Sum(n) \leqslant RecDubSum(n)$*

Proof By induction on n.

- $n \leq 1$: $Sum(n) \simeq \mathsf{skip} \simeq RecDubSum(n)$
- $n > 1$:

$$\begin{array}{lll} & Sum(n) & \\ \simeq & & \text{Def. } Sum, \lfloor n/2 \rfloor \geq 1 \\ & \underbrace{add; \ldots; add}_{\lfloor n/2 \rfloor}; Sum(\lceil n/2 \rceil) & \\ \leqslant & & \text{Corollary 4.4.14.1} \\ & add^{\lfloor n/2 \rfloor}; Sum(\lceil n/2 \rceil) & \\ \leqslant & & \text{Induction Hypothesis} \\ & add^{\lfloor n/2 \rfloor}; RecDubSum(\lceil n/2 \rceil) & \\ \simeq & & \text{Def. } RecDubSum \\ & RecDubSum(n) & \end{array}$$

□

Hence we arrive at the following refinement ordering (for $\#M = n$):

$$\langle Sum(n), M \rangle \;\leqslant\; \langle RecDubSum(n), M \rangle \;\leqslant\; \langle add^{n-1}, M \rangle \;\precapprox\; \langle \Gamma_{add}, M \rangle$$

7.1.2 Concluding Remarks

Because of its relative simplicity, the summation case provides a clear introductory example of the method of program development we propose. Besides its simplicity, there are some other reasons for including this case.

- Summation is a special instance of a *reduce*-style computation [16]. The results we have proven here for summation carry over straightforwardly to this widely used class of reducer computations.

- The case of solving triangular systems of linear equations (in Section 7.5) will illustrate the ubiquity of reducer-style computations. There, summation forms a sub-computation of a more complex solution method. The refinements that we have proven here can be reused in that case.

- Although admittedly small, our framework is the first to show the hierarchy of summation schedules. This hierarchy shows that sequential summation is a special case of recursive doubling.

Figure 7.1 depicts the derivation trajectory of the refinements that were derived in this section.

Figure 7.1: Lattice of Refinements for Summation Schedules

7.2 Prime Sieving

A natural number k is *prime* if there are no numbers other than 1 and k such that k is a multiple of that number. Formally,

$$prime(k) \Leftrightarrow \forall i : 2 \leq i \leq \lfloor \sqrt{k} \rfloor : k \bmod i \neq 0$$

Any number $k > 1$ that is non-prime is called *composite* .

In this section we address the problem of determining for all elements of a set $\{2, \ldots, N\}$ of natural numbers (with $N > 1$), whether or not they are prime.

The above characterization yields a constructive method for verifying whether a number is prime: for all numbers between 1 and k we check whether they divide k. If there is one number that divides k, then we conclude that k is non-prime.

7.2.1 A Gamma Program for Prime Sieving

For our Gamma program we choose to represent the input by the multiset $M_0 = \{2, 3, \ldots, N\}$. The program for computing prime numbers consists of a single rewrite rule called *sieve*:

$$sieve = c, d \mapsto d \Leftarrow c \bmod d = 0$$

This program repeatedly selects two numbers c and d from the multiset. If c is a multiple of d, then c is removed from the multiset. The program terminates when no numbers can be found in the multiset such that one is a divisor of the other. Hence all remaining numbers are prime.

The correctness of this Gamma program is established by construction from specification in [12] and by a posteriori verification in [79]. The correctness proof in [79] shows that the following property holds at termination of the program

$$\begin{aligned} TS \quad &\Leftrightarrow \quad \forall x : 2 \leq x \leq N : prime(x) \\ \text{or, equivalently} \quad & \\ &\qquad \forall x, y : 2 \leq x \leq N, 2 \leq y \leq \lfloor \sqrt{x} \rfloor : x \bmod y \neq 0 \end{aligned} \tag{7.1}$$

7.2.2 The Most General Schedule and a First Refinement

The most general schedule for the Gamma program *sieve* is

$$S \mathrel{\widehat{=}} !(sieve \rightarrow S) \tag{7.2}$$

From the initialisation and the fact that the *sieve* program never inserts numbers that were not already present in the multiset, follows that, during execution of the *sieve* program, all numbers in the multiset fall within the interval $[2, N]$. This statement is formalized by Lemma 7.2.1.

Lemma 7.2.1 *invariant* $\forall x : 2 \leq x \leq N$

Proof Straightforward. □

This invariant property can be used to specialize the enabling condition of the rewrite rule *sieve* such that it explicitly dictates the interval from which c and d must be taken. In addition to the above invariant, we incorporate the knowledge that if a number c is a composite, then it has a divisor d in the interval $2 \leq d \leq \lfloor \sqrt{c} \rfloor$. To this end, we define the predicate $dom(c, d)$ by

$$dom(c, d) \Leftrightarrow 2 \leq c \leq N \ \wedge \ 2 \leq d \leq \lfloor \sqrt{c} \rfloor$$

Then $sieve'$, defined below, is a strengthening of *sieve*.

$$sieve' = c, d \mapsto d \Leftarrow c \bmod d = 0 \ \wedge \ dom(c, d)$$

Let S' be the schedule S where *sieve* is replaced by the strengthening $sieve'$.

$$S' \mathrel{\widehat{=}} !(sieve' \rightarrow S') \tag{7.3}$$

By Corollary 6.2.2 follows $S' \leqslant^{\diamond}_{M_0} S$.

As a next step in the derivation, we decompose the domain of the variables of the rewrite rule $sieve'$. There are two alternatives: decomposing the domain of c and decomposing the domain of d. We consider these alternatives in turn.

Decomposing the Interval of Composites

In executing the rewrite rule $sieve'$, the variable c is to be matched with a composite number. Hence c may be matched with any number in the interval $[2, N]$. By creating a strengthening of $sieve'$ for every number in this interval, we can control the order in which the different numbers are tested for primality by scheduling these strengthenings.

The decomposition of the single rule $sieve'$ into a collection of rules where there is one for every possible value of c effectively decomposes the task of computing the primes

in the interval $[2, N]$ into $N-1$ tasks, each of which computes for exactly one number in this interval whether or not it is prime. These $N-1$ tasks are independent, hence can be executed in parallel.

We introduce a collection of schedules which contains a strengthening $sieve_k$ of the rule $sieve'$ for every value of $k : 2 \leq k \leq N$.

$$\begin{array}{lcl} sieve_k & \widehat{=} & c, d \mapsto d \Leftarrow c \bmod d = 0 \ \wedge \ dom(c,d) \ \wedge \ c = k \\ S_k & \widehat{=} & !(sieve_k \rightarrow S_k) \end{array}$$

From Lemma 3.3.31 follows that at termination of S_k holds

$$TS_k \Leftrightarrow \forall x, y : dom(x,y) \ \wedge \ x = k : x \bmod y \neq 0$$

Informally: there are no multiples of k in the multiset.

Next, we show that the result achieved at termination of S_k can not be invalidated by the *sieve* program.

Lemma 7.2.2 $\forall k : 2 \leq k \leq N : \ stable\ TS_k$

Proof The termination predicate TS_k states that there are no divisors of k in the multiset. Execution of *sieve* does not insert any new elements in the multiset. Hence, this property holds after execution of *sieve*. □

Because the different schedules S_k do not interfere with each other, we may refine S' by the parallel composition of all S_k's. Define

$$S'' \widehat{=} \Pi_{k=2}^{N} S_k \tag{7.4}$$

By Lemma 6.2.8 follows $S'' \lesssim^{\diamond}_{M_0} S'$.

We proceed by decomposing the domain of d in every component S_k. Every component S_k is responsible for determining whether the number k is prime or non-prime. To this end, the rewrite rule $sieve_k$ tries to divide k by some natural number. The value of potential divisors is matched with the variable d. If k is non-prime, then it has a divisor in the interval $[2, \lfloor\sqrt{k}\rfloor]$. For all possible values of d in this interval, we introduce

a strengthening and an encompassing schedule. Define, for all $l : 2 \leq l \leq \lfloor\sqrt{k}\rfloor$,

$$\begin{array}{lcl} sieve_{k,l} & \widehat{=} & c, d \mapsto d \Leftarrow c \, mod \, d = 0 \ \wedge \ c = k \ \wedge \ d = l \\ S_{k,l} & \widehat{=} & !(sieve_{k,l} \rightarrow S_{k,l}) \end{array} \tag{7.5}$$

Note that through the order in which we decompose (first c, then d), we can restrict the domain of d by the upper bound $\lfloor\sqrt{k}\rfloor$.

By Lemma 3.3.31 follows that $TS_{k,l}$, defined below, holds at termination of $S_{k,l}$. Informally, $TS_{k,l}$ says that if k and l are present in the multiset, then k is not a multiple of l.

$$TS_{k,l} \Leftrightarrow \forall x, y : x = k \ \wedge \ y = l : x \, mod \, y \neq 0$$

The next lemma shows that the results obtained by the individual components $S_{k,l}$ are stable.

Lemma 7.2.3 $\forall k, l : dom(k, l) : \ stable \ TS_{k,l}$

Proof Suppose $TS_{k,l}$ holds. Then either k and l are present and l does not divide k. This also holds after execution of *sieve*. Alternatively, at least one of k and l is absent from the multiset. This also holds after execution of *sieve* because this rule can only remove elements from the multiset. □

Analogous to the previous refinement, we may refine every component S_k by the parallel composition of the corresponding collection of strengthened schedules. Define

$$T_k \widehat{=} \Pi_{l=2}^{\lfloor\sqrt{k}\rfloor} S_{k,l} \tag{7.6}$$

From Lemma 6.2.8 follows that $T_k \lesssim^{\diamond}_{M_0} S_k$.

By compositionality of weak convex refinement we may refine S'' by substituting T_k for S_k. Formally, let

$$T \widehat{=} \Pi_{k=2}^{N}(\Pi_{l=2}^{\lfloor\sqrt{k}\rfloor} S_{k,l}) \tag{7.7}$$

Then, $T \lesssim^{\diamond}_{M_0} S''$.

By commutativity of ' $\|$ ' follows that T can be written equivalently as

$$T \widehat{=} \Pi_{2 \leq k \leq N, \, 2 \leq l \leq \lfloor\sqrt{k}\rfloor} S_{k,l} \tag{7.8}$$

A final refinement along this direction is justified by the observation that $sieve_{k,l}$

disables itself; i.e. if a rule $sieve_{k,l}$ is executed (successfully or failing), it establishes $TS_{k,l}$. And if $TS_{k,l}$ holds, then execution of $sieve_{k,l}$ fails. Since $TS_{k,l}$ is stable, this ensures that henceforth $sieve_{k,l}$ can never execute successfully. By Lemma 6.2.24 follows $sieve_{k,l} \lesssim^{\diamond}_{M_0} S_{k,l}$.

The schedule that is obtained by replacing $S_{k,l}$ by $sieve_{k,l}$ in T is defined by

$$T' \mathrel{\widehat{=}} \Pi_{2\leq k\leq N,\, 2\leq l\leq \lfloor\sqrt{k}\rfloor} sieve_{k,l} \tag{7.9}$$

By compositionality of weak convex refinement follows $T' \lesssim^{\diamond}_{M_0} T$.

Several further refinements can be derived by scheduling the individual rewrites $sieve_{k,l}$ of T' in particular sequential orderings. The introduction of sequential ordering can be thought of as the introduction of synchronization. This can be done in such a way that the resulting behaviour matches the characteristics of a particular architecture. For instance, define

$$\begin{array}{lcl} T'' & \mathrel{\widehat{=}} & \Pi_{2\leq k\leq N} T''_k(2) \\ T''_k(l) & \mathrel{\widehat{=}} & l \leq \lfloor\sqrt{k}\rfloor \triangleright sieve_{k,l}; T''_k(l+1) \end{array}$$

and

$$\begin{array}{lcl} T'''(k) & \mathrel{\widehat{=}} & k \leq N \triangleright T'''_k; T'''(k+1) \\ T'''_k & \mathrel{\widehat{=}} & \Pi_{2\leq l\leq \lfloor\sqrt{k}\rfloor} sieve_{k,l} \end{array}$$

Using Corollary 4.4.14.1 it is straightforward to show $T'' \leqslant T'$ and $T'''(2) \leqslant T'$.

The schedules T'' and $T'''(2)$ differ with respect to the amount of work done between synchronizations. This is also called *grain-size*. A large grain-size (here T'') is better suitable for MIMD systems and a small grain-size ($T'''(2)$) is better suitable for SIMD systems.

Decomposing the Interval of Divisors

In the previous section we investigated the method of refining the schedule S' (7.3) by first decomposing the interval of composites (the possible values of the variable c). In this section we investigate the alternative, i.e. we proceed from S' by first decomposing the interval of divisors which is ranged over by the variable d.

From Lemma 7.2.1 follows that the (composites) variable c can be matched with any value from the interval $[2, N]$. In order to check primality of any number in this interval, it suffices to check whether it is a multiple of some number in the range $[2, \lfloor\sqrt{N}\rfloor]$. This

range constitutes the interval over which the (factors) variable d needs to range.

For every possible value in the interval $[2, \lfloor\sqrt{N}\rfloor]$ we introduce a strengthening $sieve'_l$ of $sieve'$ and an encompassing schedule. Define, for all $l : 2 \leq l \leq \lfloor\sqrt{N}\rfloor$

$$\begin{array}{lcl} sieve'_l & \widehat{=} & c, d \mapsto d \Leftarrow c \bmod d = 0 \ \wedge \ dom(c,d) \ \wedge \ d = l \\ S'_l & \widehat{=} & !(sieve'_l \rightarrow S'_l) \end{array}$$

Execution of a schedule S'_l removes all multiples of l in the interval $[2, N]$ from the multiset. This is described formally by the termination predicate TS'_l. By Lemma 3.3.31 TS'_l holds at termination of S'_l.

$$TS'_l \Leftrightarrow \forall x, y : dom(x,y) \ \wedge \ y = l : x \bmod y \neq 0$$

Next, we show that the result achieved at termination of S'_l can not be invalidated by the program *sieve*, hence it can not be invalidated by any of the other schedules S'_l.

Lemma 7.2.4 $\forall l : 2 \leq l \leq \lfloor\sqrt{N}\rfloor : \ stable\ TS'_l$

Proof If TS'_l holds, then there are no multiples of l in the multiset. Execution of *sieve* only removes elements from the multiset, hence it can never invalidate this property. □

Let U be the parallel composition of all components S'_l

$$U \widehat{=} \Pi_{l=2}^{\lfloor\sqrt{N}\rfloor} S'_l \tag{7.10}$$

Then, by Lemma 6.2.8 follows $U \lesssim^{\diamond}_{M_0} S'$.

A subsequent refinement may be obtained by decomposing the domain of c; i.e. by creating specific instances of the rewrite rules $sieve'_l$ for all possible values of c. In contrast to the second domain decomposition in the previous section, we can, using the current order of domain decomposition, not derive an upper bound on the composite variable c that depends on the value of (the divisor) l.

For this decomposition, we can use the schedules $S_{k,l}$ that we introduced before (7.5). By Lemma 7.2.3 follows that the different schedules $S_{k,l}$ do not interfere with each other, hence by Lemma 6.2.8 follows $\Pi_{2 \leq k \leq N} S_{k,l} \lesssim^{\diamond}_{M_0} S'_l$ (for all l). By compositionality of weak convex refinement, we obtain a refinement of U by replacing each subterm S'_l of U by

$\Pi_{2\leq k\leq N} S_{k,l}$. Formally:

$$U' \lesssim^{\diamond}_{M_0} U \quad \text{where} \quad U' \mathrel{\widehat{=}} \Pi_{2\leq k\leq N, 2\leq l\leq \lfloor\sqrt{N}\rfloor} S_{k,l}$$

Because execution of a rewrite rule $sieve_{k,l}$ disables itself, we obtain, analogous to the previous derivation, a refinement of U' by replacing $S_{k,l}$ by $sieve_{k,l}$. Formally:

$$U'' \lesssim^{\diamond}_{M_0} U' \quad \text{where} \quad U'' \mathrel{\widehat{=}} \Pi_{2\leq k\leq N, 2\leq l\leq \lfloor\sqrt{N}\rfloor} sieve_{k,l}$$

The schedule U'' performs more rewrites than the schedule T' (7.9) that we ended with in the previous derivation. However, some of the rewrites that the parallel strategies U, U' and U'' perform may be omitted if we introduce a sequential ordering on the components we have derived thus far. To illustrate this, we proceed with an alternative avenue of refinement (starting from U (7.10)).

A sequential order of executing S'_l's is described by the schedule $E(2, \lfloor\sqrt{N}\rfloor)$ where $E(i, ub)$ is defined by

$$E(i, ub) = i \leq ub \triangleright (S'_l; E(i+1, ub))$$

Lemma 7.2.5 shows that $E(2, \lfloor\sqrt{N}\rfloor)$ is a (stateless) refinement of U.

Lemma 7.2.5 $E(2, \lfloor\sqrt{N}\rfloor) \leqslant U$

Proof Straightforward induction proof using Corollary 4.4.14.1. □

Now, consider the multiset after termination of S'_2. Then there are no strict multiples of 2 left in the multiset. After termination of S'_3 there are no strict multiples of 3 left. And, in general, after termination of S'_l, there are no strict multiples of l left. Formally:

Lemma 7.2.6 $\forall M : M \in \mathcal{O}^{\diamond}(S'_2; S'_3; \ldots; S'_l, M_0) : (\forall i : 2 \leq i \leq l : [\![TS'_i]\!] M)$

Proof By Lemma 3.3.31 and Lemma 7.2.4. □

The fact that after termination of S'_2 all multiples of 2 have been eliminated, causes the subsequent rewrite rules $sieve'_l$ where l is a multiple of 2 (i.e. $4, 6, 8, \ldots$) to fail after having searched the multiset exhaustively for an enabling pair of elements. Hence, removing these rewrite rules from the schedule avoids this superfluous search. Formally,

$$TS'_2 \Leftrightarrow (\forall i : 2 \leq i \leq \frac{N}{2} : \dagger sieve'_{2*i})$$

Because TS'_2 is stable we get, by Lemma 6.2.23, for all M such that $[\![TS'_2]\!]M$,

$$\forall i : 2 \leq i \leq \frac{N}{2} : \mathsf{skip} \lesssim^{\diamond}_{M} S'_{2*i} \tag{7.11}$$

Hence S'_l can be omitted for all even values of l. This idea is incorporated in the schedule $E'(2, \lfloor\sqrt{N}\rfloor)$ which is defined as follows

$$\begin{array}{lcl} E'(i, ub) & \widehat{=} & i \leq ub \triangleright (\; i = 2 \triangleright \quad (S'_2; E'(3, ub)) \\ & & \qquad\qquad\quad [S'_i; E'(i+2, ub)] \;) \end{array}$$

By (7.11) and Theorem 6.2.19 follows $E'(2, \lfloor\sqrt{N}\rfloor) \lesssim^{\diamond}_{M_0} E(2, \lfloor\sqrt{N}\rfloor)$.

A similar argument can be made after termination of S''_3. And subsequently for S''_5 and S''_7 and so on. However, the indices of these components are exactly the prime numbers that we are looking for. Therefore we will not use them any further in the construction of the method for computing them.

We proceed with eliminating superfluous rewrites in the components S''_l of E''. To this end, we decompose the rewrite rules $sieve''_l$ with respect to the possible values of variable c. When S_l is scheduled for execution, all multiples of numbers $2, \ldots, l-1$ have been removed from the multiset. Hence, the variable c can only be matched to odd numbers that are multiples of l. The first of these is $l * l$ (where l is odd) and the subsequent numbers are obtained by adding an even number of l's. This series of numbers is defined by $e_{l,k} = l * l + k * (2 * l)$ for odd numbers $l \geq 3$ and $k \geq 0$.[1]

We introduce the following strengthenings and their corresponding schedules, for all $l, k : odd(l) \; \wedge \; l^2 \leq e_{l,k} \leq N$

$$\begin{array}{lcl} sieve''_{l,k} & \widehat{=} & c, d \mapsto d \Leftarrow c \bmod d = 0 \; \wedge \; d = l \; \wedge \; c = e_{l,k} \\ F_{l,k} & \widehat{=} & !(sieve''_{l,k} \rightarrow F_{l,k}) \end{array}$$

At termination of $F_{l,k}$ holds

$$TF_{l,k} = \forall x, y : x = k \; \wedge \; y = l : x \bmod y \neq 0$$

Analogous to Lemma 7.2.4, follows that $TF_{l,k}$ is stable. Define F'_l as the parallel com-

[1] In the schedule E'' the components S''_l where $l > 2$ and l is even have been removed. Hence if $l > 2$ is the index of a component, then l must be odd. If furthermore m is odd, then $l*l + m*l = (l+m)*l$ is even, hence $e_{l,k}$ is odd.

position of all $F_{l,k}$'s:

$$F'_l \mathrel{\widehat{=}} \Pi_{k:l^2 \leq e_{l,k} \leq N} F_{l,k}$$

Hence, by Lemma 6.2.8, follows for all $l \geq 3$ that if $\forall i : 2 \leq i < l : [\![TS_i]\!]M$, then $F'_l \lesssim^{\diamond}_M S''_l$.

The decomposition of S'_2 is handled as a special case. Let $e_{2,k} = 4 + 2 * k$ for $k \geq 0$ and define, for all $k : 4 \leq e_{2,k} \leq N$,

$$\begin{array}{lcl} sieve''_{2,k} & \mathrel{\widehat{=}} & c, d \mapsto d \Leftarrow c \bmod d = 0 \ \wedge \ d = 2 \ \wedge \ c = e_{2,k} \\ F_{2,k} & \mathrel{\widehat{=}} & !(sieve''_{2,k} \rightarrow F_{2,k}) \end{array}$$

At termination of $F_{2,k}$ holds

$$TF_{2,k} = \forall x, y : x = k \ \wedge \ y = 2 : x \bmod y \neq 0$$

Analogous to Lemma 7.2.4, follows that $TF_{2,k}$ is stable. Define

$$F'_2 \mathrel{\widehat{=}} \Pi_{k:4 \leq e_{2,k} \leq N} F_{2,k}$$

Then, by Lemma 6.2.8, follows $F'_2 \lesssim^{\diamond} S'_2$.

Clearly, $sieve''_{l,k}$ disables its own execution for any $l \geq 2$ and $k \geq 0$. Hence by Lemma 6.2.24 follows

$$sieve''_{l,k} \lesssim^{\diamond}_{M_0} F_{l,k}$$

Using these results, we may refine $E''(2, \lfloor\sqrt{N}\rfloor) \lesssim^{\diamond}_{M_0} E'(2, \lfloor\sqrt{N}\rfloor)$ where $E''(i, ub)$ is defined by

$$\begin{array}{lcll} E''(i, ub) & \mathrel{\widehat{=}} & i \leq ub \triangleright (\ i = 2 \triangleright & (\Pi_{k:4 \leq e_{2,k} \leq N} \ sieve''_{2,k}; E''(3, ub)) \\ & & & [\Pi_{k:i^2 \leq e_{i,k} \leq N} \ sieve''_{i,k}; E''(i+2, ub)] \ ) \end{array}$$

Schedule E'' is a parallel variant of the prime sieving algorithm which was discovered by Eratosthenes in about 240 BC. His algorithm is still considered to be the most efficient for computing small prime numbers.

7.2.3 Concluding Remarks

During the derivation of the different coordination strategies in this section we ran into a common trade-off in algorithm design: in sequential execution, mathematical ingenuity

may lead to the identification of unnecessary computations, which can be omitted. On the other hand, parallel execution requires more computations than sequential execution, but may require less time to execute. It depends on the speed-up gained by parallel execution whether such a trade is worthwhile.

We briefly describe some similarities between our method for deriving coordination strategies and the Dijkstra-Gries approach (see [47], [61], [50]) to program development.

The Dijkstra-Gries approach consists of the structured transformation of a specification in predicate logic into program fragments. A typical step in their method is the replacement of a universally quantified expression, say $\forall i : l \leq i \leq h : p(i, \ldots)$, into a (`for`-)loop structure where a variable consecutively takes on all values from the range of quantification; e.g. for the given example:

$$\texttt{for } i = l \texttt{ to } h \texttt{ do} \text{ establish } p(i, \ldots)$$

In our method of refinement, a very similar step, called "decomposition", is used. A rewrite rule which has to match some variable j which may range over an interval $j : l' \leq j \leq h'$ may be replaced by the composition of a collection of strengthened rewrite rules where there is exactly one strengthening for every possible value for j.

A significant difference, however, is that the Dijkstra-Gries approach suggests the introduction of sequential-loop structures while our method of refinement leaves the order in which the range of the quantification is to be traversed open and thereby leaves this up to the program designer to fill in.

Figure 7.2 depicts the method by which the refinements are related.

A classical exposition of a formal derivation of Eratosthenes prime sieving algorithm is [74]. The prime-sieving problem was used by several authors (e.g. Gries et al. [62]) to show that formal derivations could guide the way to new, more efficient algorithms. This resulted in the discovery of a number of prime-sieving algorithms. One of the inventors of these new algorithms illustrated the relationship between the newly discovered algorithms by means of a family-tree [101].

In [101], Pritchard observes that for the construction of his family of algorithms it was advantageous to start with a nondeterministic initial description since this allows program transformations that yield different temporal-orderings of the operations of the program. This observation agrees with the principles that underlie the method for program design presented in this thesis.

Figure 7.2: Lattice of Refinements of Prime Sieving Schedules

> *Indeed, I believe that virtually* every *important aspect of programming arises somewhere in the context of sorting and searching!*
>
> – Donald Knuth [81], p. v

7.3 Sorting

A classical problem in Computer Science is that of sorting. This problem requires that some collection of input elements is rearranged into nondecreasing order.

The sorting problem is an interesting subject of study because it is easy to understand but has many facets. Furthermore, it is a problem for which many different solutions have been proposed. In this section we will derive several of these.

We formally specify the sorting problem as follows. Recall from Definition 2.2.3 that we write $\bar{l} \downarrow k$ to denote the number of occurences of k in the sequence $\bar{l}$. Define the predicate *permutation* over pairs of sequences as follows

$$\mathit{permutation}(\bar{l}, \bar{l'}) \;\Leftrightarrow\; \forall k : k \in \bar{l} \vee k \in \bar{l'} : \bar{l} \downarrow k = \bar{l'} \downarrow k$$

If the input is some sequence $\overline{v} = \langle\, v_1, \ldots, v_N \,\rangle$, then the output a sorting program should be a sequence $\overline{v'}$ such that

$$\mathit{permutation}(\overline{v}, \overline{v'}) \tag{7.12}$$

$$\forall i, j : 1 \leq i < j \leq N : v'_i \leq v'_j \tag{7.13}$$

We model the input sequence $\overline{v}$ by the multiset $M_0 = \{(i, v_i) \mid 1 \leq i \leq N\}$ of index-key pairs. In [12], a Gamma program for sorting is formally derived. It consists of the following multiset rewrite rule

$$\mathit{swap} = (i, x), (j, y) \mapsto (i, y), (j, x) \Leftarrow i < j \;\wedge\; x > y$$

This rule essentially encodes a *compare-exchange* operation: if a key at a lower position is larger than a key at a higher position, then exchange these keys.

The compare-exchange rule does not require additional storage for auxiliary results. Therefore, any strategy based on this rule is a so-called "*in-place*" sorting method.

A correctness proof for this sorting program was given in Section 2.2. Here, we recall

some results that will be used in the derivation of coordination strategies.

All indices i for elements (i, x) are from the interval $[1, N]$.

Lemma 7.3.1 *invariant* $\forall i, x : (i, x) : 1 \leq i \leq N$

Proof Straightforward from the definition of *swap*. □

Lemma 7.3.2 shows that there is always exactly one element in the multiset that represents the i^{th} element from the sequence. This allows us to write x_i for x from (x, i).

Lemma 7.3.2 *invariant* $\forall i : 1 \leq i \leq N : \#(i, x) = 1$

Proof Straightforward. □

Similarly, we can show that at any stage during execution, the key-values of the elements in the multiset contain the values of the original input sequence.

Lemma 7.3.3 *invariant* $\forall i, x : (i, x) : x \in \overline{v}$

Proof Straightforward from the definition of *swap*. □

7.3.1 The Most General Schedule and a First Refinement

We start the derivation of coordination strategies by constructing the most general schedule for the sorting program *swap*.

$$S \mathrel{\widehat{=}} !(swap \rightarrow S) \tag{7.14}$$

Invariants 7.3.1 and 7.3.3 indicate the domains over which the variables in the rewrite rule *swap* range. This information is encoded in the rewrite rule *swap*′:

$$\begin{array}{lcl} dom(x, y) & \Leftrightarrow & x, y \in \overline{v} \\ swap' & = & (i, x), (j, y) \mapsto (i, y), (j, x) \Leftarrow 1 \leq i < j \leq N \wedge x > y \wedge dom(x, y) \\ S' & \mathrel{\widehat{=}} & !(swap' \rightarrow S') \end{array} \tag{7.15}$$

Then by Lemma 6.2.1 follows $S' =^{\diamond}_{M_0} S$.

A reason for specifying the domains of the variables explicitly, is that this may indicate directions for refinement: if the domain over which a variable ranges is finite, then it may be worthwile to investigate whether the decomposition of the schedule with respect to this variable (using laws from Section 6.2.2) yields an interesting avenue for refinement. This refinement strategy was illustrated by the prime sieving case study in Section 7.2. For the sorting problem, we will investigate this approach in Section 7.3.4. First, we will examine some coordination strategy for the sorting program whose correctness is proven using simulation-based techniques.

7.3.2 BubbleSort

In this section we consider the Bubble Sort algorithm. Descriptions of this algorithm can be found in [3], [81]. A derivation of Bubble Sort using the Dijkstra-Gries approach is described in [80].

The Bubble Sort strategy for sorting is described by the schedule $BS(1, N)$ which is defined by

$$\begin{array}{lcll} BS(n,m) & \widehat{=} & m > n \triangleright & swap_m; BS(n, m-1) \\ & & & [n < N \triangleright BS(n+1, N)] \end{array}$$

where

$$swap_m \quad = \quad (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow i < j \wedge i = m-1 \wedge j = m \wedge x > y$$

We will show that Bubble Sort is a correct coordination strategy for the sorting program *swap*. To this end, we prove that the schedule $BS(1, N)$ is a convex refinement of the most general schedule.

Note that all schedules of $\langle BS(1,n), M_0\rangle$-derived configurations are of the form $BS(n, m)$. We introduce the following predicates for describing $\langle BS(1,n), M_0\rangle$-derived configurations.

$$\begin{array}{lcl} Idx(n,m) & \Leftrightarrow & 1 \leq n \leq m \leq N \\ Ord(n) & \Leftrightarrow & \forall i, x : (i,x) : 1 \leq i < n : (\forall j, y : (j,y) : i < j \leq N : x \leq y) \\ Min(m) & \Leftrightarrow & \exists z : (m,z) : z = (min\ i, x : (i,x) : m \leq i \leq N : x) \end{array}$$

The quantified predicate $Ord(n)$ states that every key at a position before n is smaller than or equal to all keys at higher positions. Hence n divides the sequence into two intervals:

- the interval $[1, n-1]$ where the keys are in sorted order, and
- the interval $[n, N]$ where the ordering of the keys is unknown and therefore assumed to be unsorted.

$Ord(N)$ implies that the sequence is sorted. However, initially $Ord(0)$ holds.

The idea behind Bubble Sort is to increase the bound between the sorted and unsorted interval from 0 to N. This is achieved by swapping the minimum key of the unknown interval to position n. This implies that this key is at its sorted position. Hence the upper bound of the sorted interval can be incremented.

For a configuration $\langle BS(n,m), M\rangle$, $Min(m)$ denotes that the minimum key of the interval $[m, N]$ is stored at position m. Hence, $Min(N)$ holds invariantly.

The following property formally justifies the increment of the upper-bound of the sorted interval.

$$Ord(n) \wedge Min(n) \Leftrightarrow Ord(n+1) \tag{7.16}$$

In Lemma 7.3.8 we will use a convex simulation to prove that $BS(1, N)$ is a refinement of S'. In this convex simulation relation we want to use the relationship between the Bubble Sort schedule and the multiset as described by the predicates Ord and Min. Therefore, we need to show that these relations can not be invalidated by interference of the kind that are possible for the convex notion of refinement. To this end, we set out to show the stability of Ord and Min, starting with Ord.

Lemma 7.3.4 $\forall n : 1 \leq n \leq N :$ *stable* $Ord(n)$.

Proof Assume $[\![Ord(n)]\!]M$ and let $\sigma = \{(i,y),(j,x)\}/\{(i,x),(j,y)\}$ be a substitution for *swap*, hence $i < j$ and $x > y$. Consider the following cases

- $i < n$: From $i < j$ and $Ord(n)$ follows $x \leq y$ which contradicts $x > y$, hence this choice of σ cannot occur.
- $i \geq n$: Exchanging keys at positions $\geq n$ does not affect $Ord(n)$.

□

If Bubble Sort is considered in isolation, then $Min(m)$ adequately describes one of its invariant properties. However, in a convex simulation interference is possible by arbitrary execution of the program *swap*. This interference may modify the position of the minimum key of the unsorted interval. However, this minimum can never be moved

below the bound that separates the sorted from the unsorted interval. We weaken *Min* to *Min'* which no longer pin-points the minimum of the unsorted interval to a particular position, but bounds its position by the interval $[n, m]$.

$$Min'(n, m) \Leftrightarrow \exists k, z : n \leq k \leq m \wedge (k, z) : z \leq (min\ i, x : (i, x) : m \leq i \leq N : x) \quad (7.17)$$

Taking $m = N$ as upper bound of the interval in which the minimum of the unsorted elements is located, yields the following invariant

$$\forall n : \ invariant\ Min'(n, N) \quad (7.18)$$

Because $Min'(n, n) \Leftrightarrow Min(n)$, we can rewrite property (7.16) as

$$Ord(n) \wedge Min'(n, n) \Leftrightarrow Ord(n + 1) \quad (7.19)$$

To show that $Min'(n, m)$ continues to hold if potentially interfering *swap*'s occur, we need to consider it in conjunction with $Ord(n)$.

Lemma 7.3.5 $\forall n, m : 1 \leq n \leq m \leq N : \ stable\ Ord(n) \wedge Min'(n, m)$

Proof Stability of $Ord(n)$ follows from Lemma 7.3.4. It remains to show that $Min'(n, m)$ is stable. Let $\sigma = \{(i, y), (j, x)\}/\{(i, x), (j, y)\}$ with $i < j$ and $x > y$ be a substitution of *swap*. Consider the possible cases for i and j

- $1 \leq i < n$: From $i < j$ and $Ord(n)$ follows $x \leq y$ which contradicts $x > y$, hence this case can not occur.

- $n \leq i < j < m$: Exchanging two keys whose positions are in the interval $[n, m)$ does not change the minimum key in that interval.

- $n \leq i < m \leq j \leq N$: From $x > y$ follows that exchanging x and y may decrease the minimum of the keys with positions in the interval $[n, m)$ and increase the minimum of key with positions in the interval $[m, N]$. Hence $Min'(n, m)$ holds after the substitution σ. Exchanging keys other than the minima does not affect $Min'(n, m)$.

- $m \leq i < j \leq N$: Exchanging two keys whose positions are in the interval $[m, N]$ leaves the minimum key of that interval unchanged. Since $m \leq i < j$, the key at

m can only decrease. Hence the minimum of the keys with positions in the interval $[n, m]$ may only decrease which ensures that $Min'(n, m)$ holds after execution of σ.

□

Suppose that a sequence is sorted up to position $n-1$ and the minimum of the unsorted interval is located at a positions in the interval $[n, m]$. Then Lemma 7.3.6 shows that execution of $swap_m$ ensures that the minimum of the unsorted interval is located at some position in the interval $[n, m-1]$.

Lemma 7.3.6 *Let $[\![Min'(n, m)]\!]M$ for n, m such that $Idx(n, m)$.*
If $\langle swap_m, M\rangle \xrightarrow{\lambda} \langle \mathsf{skip}, M'\rangle$, then $[\![Min'(n, m-1)]\!]M'$.

Proof Let $(m-1, x), (m, y) \in M$ and $(m-1, x'), (m, y') \in M'$. Independent of the success or failure of $swap_m$ holds $x' = min(x, y)$. We consider the following cases

1. The minimum of the keys with positions in the interval $[n, m]$ in M is at position m. From $x' = min(x, y)$ follows that in M' the minimum is at position $m-1$. Hence $[\![Min'(n, m-1)]\!]M'$.

2. The minimum of the keys with positions in the interval $[n, m]$ in M is at some position $< m$. Then immediately $[\![Min'(n, m-1)]\!]M'$.

□

In Lemma 7.3.8 we use a weak convex simulation to show that the schedule $BS(1, N)$ refines S'. One of the proof-obligations induced by this method, is to show that S' can mimic, by zero or more transitions, every transition that the schedule $BS(n, m)$ may make. This obligation is fulfilled by the following lemma. Furthermore, this lemma shows that the schedule S' may arrive in the same form S' after mimicking a transition from $BS(n, m)$ for some n, m.

Lemma 7.3.7 *If $\langle BS(n, m), M\rangle \xrightarrow{\lambda} \langle s, M'\rangle$, then $\langle S', M\rangle \xrightarrow{\overline{\lambda'}}{}^* \langle S', M'\rangle$ such that $\overline{\lambda'} = \varepsilon^k \cdot \widehat{\lambda}$ for some $k \geq 0$.*

Proof Consider the possible cases for λ:

- $\lambda = \varepsilon$: then $M' = M$ and by definition of $\longrightarrow^*$ follows $\langle S', M\rangle \xrightarrow{\langle\ \rangle}{}^* \langle S', M\rangle$.

- $\lambda = \sigma$: Because $\mathcal{L}(BS(n,m)) \lhd \mathcal{L}(swap')$ we get by Lemma 3.3.23 and the definition of $\longrightarrow^*$ that $\langle S', M\rangle \xrightarrow{\sigma}{}^* \langle S', M'\rangle$.

□

The preceding results are used in the following lemma to prove that $BS(1,N)$ is a refinement of S' (7.15).

Lemma 7.3.8 $BS(1,N) \lesssim^{\diamond}_{M_0} S'$

Proof Let

$$\mathcal{R} = \{(\langle BS(n,m), M\rangle, \langle S', M\rangle) \mid Idx(n,m) \wedge [\![Ord(n)]\!]M \wedge [\![Min'(n,m)]\!]M\}$$

We show that $\mathcal{R}$ is a weak convex simulation. Assume $\Diamond(M, M')$.
By Lemma 7.3.5 follows $[\![Ord(n)]\!]M'$ and $[\![Min'(n,m)]\!]M'$.
transition
Assume $\langle BS(n,m), M'\rangle \xrightarrow{\lambda} \langle s', M''\rangle$. By definition of $BS(n,m)$ follows that $s' \equiv BS(n',m')$ for some n', m'. By Lemma 7.3.7 follows that $\langle S', M'\rangle \xrightarrow{\overline{\lambda}}{}^* \langle S', M''\rangle$ such that $\overline{\lambda'} = \varepsilon^k \cdot \widehat{\lambda}$ for some $k \geq 0$.

Next, we show that the predicates *Idx*, *Ord* and *Min'* hold in the new configuration. By definition of $BS(n,m)$ follows that the transition is derived from the execution of the rewrite rule $swap_m$. Then, by Lemma 7.3.6 follows $[\![Min'(n, m-1)]\!]M''$.

By Lemma 7.3.4 follows $[\![Ord(n)]\!]M''$. Consider the following cases for n and m

- $m \neq n$ and $n \neq N$: Then $s' = BS(n', m')$ where $n' = n$ and $m' = m - 1$.
  Hence $Idx(n', m')$ and $(\langle BS(n', m'), M''\rangle, \langle S', M''\rangle) \in \mathcal{R}$.

- $m = n$ and $n < N$: Then $BS(n,m) \equiv BS(n+1, N)$. Hence $s' = BS(n', m')$ where $n' = n + 1$ and $m = N - 1$. By (7.19) follows $[\![Ord(n+1)]\!]M''$.

  By (7.18) follows $[\![Min'(n+1, N)]\!]M''$. Then, by Lemma 7.3.6 follows $[\![Min'(n+1, N-1)]\!]M''$.

  Clearly $Idx(n+1, N-1)$, hence $(\langle BS(n', m'), M''\rangle, \langle S', M''\rangle) \in \mathcal{R}$.

- $m = n$ and $n = N$: then $s = BS(N-1, N-1) \equiv BS(N, N) \equiv \mathsf{skip}$.
  This contradicts the assumption that s makes a transition.

termination
If $BS(n,m) \equiv \mathsf{skip}$, then $n = m = N$. Then $[\![Ord(N)]\!]M'$ implies $[\![\dagger swap]\!]M'$. By a straightforward derivation follows $\langle S', M'\rangle \xrightarrow{\varepsilon}{}^{*} \langle \mathsf{skip}, M'\rangle$.

Finally, $(\langle BS(1,N), M_0\rangle, \langle S', M_0\rangle) \in \mathcal{R}$ follows from $Idx(1,N)$, $[\![Ord(1)]\!]M_0$ and $[\![Min'(1,N)]\!]M_0$. □

The main difference between the convex-based correctness proof for Bubble Sort in this section and usual invariants-based approaches is the weaker version Min' (of Min). Whereas Min precisely states the position of some minimum key, Min' approximates this position by bounding it within an interval.

The added value gained by showing that $BS(1,N)$ is a *convex* refinement, is that this ensures that the schedule yields the correct output even if it is executed in an environment where other processes are executing *swap*'s on the same data-space. In particular, $BS(1,N)$ put in parallel with itself yields the correct output. By Lemma 4.4.27 follows that executing an arbitrary number of copies of $BS(1,N)$ in parallel will produce the correct output.

Recall that convex refinement is a precongruence, hence gives rise to a number of algebraic laws that may be used in a modular fashion. One might wonder if it would be simpler to derive a Bubble Sort coordination strategy in an algebraic style using the laws for convex refinement. This question is answered in the next section.

7.3.3 Ripple Sort

In Chapters 4 and 6 we have stressed the importance of modular equational reasoning and hence the prerequisite property of precongruence of the notion of refinement. In this section we show that even though statebased refinement is not a precongruence, it can be effectively used to reason about refinement.

We present a new coordination strategy called *Ripple Sort* and use statebased simulation to show that it is an intermediate strategy between the most general schedule and Bubble Sort.

Sorting methods that operate by swapping only neighbouring elements (implicitly) assume that the indices of the elements to be sorted constitute a contiguous interval of integers. The Gamma program *swap* does not depend on this assumption: it also sorts sets of data that do not have contiguous (integer) indices; e.g. an initial multiset $\{(2,B),(5,C),(8,A)\}$ is sorted into $\{(2,A),(5,B),(8,C)\}$.

However, if the data to be sorted is modelled as a sequence with consecutive indices, as we have done with the initial multiset M_0, then the Gamma program *swap* maintains this property throughout execution (*cf.* the invariant of Lemma 7.3.2).

We proceed by showing how the assumption that the data to be sorted is modelled as a sequence with consecutive indices may be used to strengthen the rewrite rule *swap* such that it compares only neighbouring indices. To this end, we first show that, under this assumption, the following characterizations of orderedness are equivalent.

(1) $\forall i : 1 \leq i < N : (\forall j : i < j \leq N : v_i \leq v_j)$

(2) $\forall i : 1 \leq i < N : v_i \leq v_{i+1}$

Characterization (2) depends on the fact that keys are numbered with consecutive indices.

Lemma 7.3.9 *If the elements to be sorted are arranged as a sequence* $\overline{v} = \langle\, v_1, \ldots, v_N \,\rangle$ *of consecutively numbered keys, then the characterizations (1) and (2) are equivalent.*

Proof

- (1) $\Rightarrow$ (2) : Immediate.
- (2) $\Rightarrow$ (1) : By induction on N. Because the indices are consecutive numbers, we can use transitivity of $\leq$.

□

The condition of the rewrite rule *swap′* corresponds to the negation of characterization (1). By Lemma 7.3.9 follows that $\neg(1) \Leftrightarrow \neg(2)$. Hence by Lemmas 7.3.1 and 6.2.1, we may replace this enabling condition of *swap′* with the negation of characterization (2) to obtain a schedule S'', defined by (7.20), such that $S'' =^{\diamond}_{M_0} S'$.

$$\begin{array}{lcl} \mathit{swap}'' & = & (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow 1 \leq i < N \wedge j = i+1 \wedge x > y \\ S'' & \widehat{=} & !(\mathit{swap}'' \rightarrow S'') \end{array} \tag{7.20}$$

The refinement we consider next, resembles a decomposition of the range of the variable i of the rewrite rule *swap″*: we introduce a rewrite rule swap'_i for every possible value of the variable i and embed these in a suitable control structure.

Define, for all $i : 1 \leq i < N$,

$$swap'_i = (i, x), (j, y) \mapsto (i, y), (j, x) \Leftarrow 1 \leq i < N \wedge j = i + 1 \wedge x > y$$

In the decomposition refinement in the prime sieving case in Section 7.2, the specialized rewrite rules could not interfere with each other and could therefore be scheduled in arbitrary order. In this case, the strengthened rewrite rules do interfere: $swap'_i$ and $swap'_{i+1}$ may both modify the element with index i. Therefore, a schedule needs to be used that takes into account that execution of one rewrite rule may enable another.

This idea leads to the following schedule, called "Ripple Sort". It is constructed so that a new rewrite rule is scheduled only if it may have been enabled by the execution of a preceding rewrite rule (or if it could be enabled initially).

$$\begin{array}{lcl} R & \widehat{=} & \Pi_{i=1}^{N-1} R_i \\ R_i & \widehat{=} & swap'_i \rightarrow (R_{i-1} \parallel R_{i+1}) \end{array} \tag{7.21}$$

To aid the intuition, we informally explain Ripple Sort's mode of operation. The schedule R spawns $N - 1$ threads R_i; one for each position $i : 1 \leq i < N$. Initially R_i detects whether the keys at positions i and $i + 1$ are in the proper relative order. If this is not the case, then a rewrite $swap'_i$ is executed to establish local orderedness. This successful execution of $swap'_i$ may invalidate the orderedness at positions $i - 1$ and $i + 1$. Therefore the thread splits into two: one for position $i - 1$ and one for $i + 1$. As before, these threads check if these positions are properly ordered (with respect to their neighbours). If this is the case, then they terminate. Otherwise, $swap'_{i-1}$ and/or $swap'_{i+1}$ are executed, and the strategy is applied recursively.

The method derives its name from the resemblance between the way that threads move outward from the position in the sequence where a swap was performed and the way ripples move outward from the place where a pebble is thrown into water.

The refinement of S by R is not of the kind that is supported by any of the convex decomposition laws. Therefore, we will resort to proving this refinement using statebased techniques.

The general form that the schedule R takes during execution is $\Pi_{i=0}^{N} R_i^{a_i}$ with $a_i \geq 0$. Next, we define a predicate, F, which relates the general form of the Ripple Sort schedule to the contents of the multiset (*cf.* the S-derived configuration property μ of Definition 3.3.5):

Definition 7.3.10 *$F(s) \Leftrightarrow \exists a_0, \ldots, a_N$ such that*

1. $s \equiv \Pi_{i=0}^{N} R_i^{a_i}$
2. $\forall i, x, y : (i, x), (i+1, y) : 0 \leq i \leq N : a_i = 0 \Rightarrow x \leq y$

Read "backwards", predicate F states that if two neighbouring keys are unordered (i.e. $x > y$), then ($a_i > 0$) there is a thread R_i in the schedule R that will order these keys. The formal proof that Ripple Sort refines the schedule S'' depends on the invariance of predicate F. This is shown by the following lemma.

Lemma 7.3.11 *If $[\![F(s)]\!]M$ and $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$, then $[\![F(s')]\!]M'$.*

Proof By Lemma 3.2.5 follows that there exists a sequence

$$\langle s_0, M_0\rangle \xrightarrow{\lambda_1}_1 \langle s_1, M_1\rangle \ldots \xrightarrow{\lambda_i}_1 \ldots \langle s_{n-1}, M_{n-1}\rangle \xrightarrow{\lambda_n}_1 \langle s_n, M_n\rangle$$

where $\langle s_0, M_0\rangle = \langle s, M\rangle$ and $\langle s_n, M_n\rangle = \langle s', M'\rangle$.

We prove that the proposition holds for single-step transitions. The result then follows by induction on the length of the transition sequence.
From $[\![F(s)]\!]M$ follows that $s \equiv \Pi_{i=0}^{N} R_i^{a_i}$ with $a_i \geq 0$. A single-step transition for s is derived, by (N2), from $\langle R_i, M\rangle \xrightarrow{\lambda} \langle t, M'\rangle$ (for some i such that $a_i > 0$), hence $s' \equiv t \parallel \Pi_{i=0}^{N} R_i^{a'_i}$ with $a'_i \geq 0$ for all i. The latter transition is in turn derived, by (N0) or (N1), from $\langle swap'_i \rightarrow (R_{i-1} \parallel R_{i+1}), M\rangle \xrightarrow{\lambda} \langle t, M'\rangle$. Both the successful and failing execution of $swap'_i$ establishes $[\![\, \forall i', x', y' : i' = i \wedge (i', x'), (i'+1, y') : x' \leq y' \,]\!]M'$.

We show $[\![F(s')]\!]M'$ by considering the following cases for λ.

- $\lambda = \varepsilon$: Then $t \equiv \mathsf{skip}$ hence $s' = \Pi_{j=0}^{N} R_j^{a'_j}$ with $a'_j = a_j$ for all $0 \leq j \leq N : j \neq i$ and $a'_i = a_i - 1$. From $[\![F(R')]\!]M$ and $x' \leq y'$ follows $[\![F(R'')]\!]M'$.
- $\lambda = \sigma$: Then $t \equiv R_{i-1} \parallel R_{i+1}$ hence $s' = \Pi_{j=0}^{N} R_j^{a'_j}$ with $a'_j = a_j$ for all $j : 0 \leq j < i-1 \vee \imath + 1 < j \leq N$, and $a'_{i-1} = a_{i-1} + 1, a'_i = a_i - 1$ and $a'_{i+1} = a_{i+1} + 1$.
  From $[\![F(s)]\!]M$ and $x' \leq y'$ follows $[\![F(s')]\!]M'$.

□

Next, we prove that Ripple Sort is a refinement of the (strengthened) most general schedule S''.

Lemma 7.3.12 $R \leqq_{M_0} S''$

Proof Let

$$\mathcal{R} = \{(\langle s, M\rangle, \langle S''^k, M\rangle) \mid [\![F(s)]\!]M, k \geq 0\}$$

Note that $(\langle R, M_0\rangle, \langle S'', M_0\rangle) \in \mathcal{R}$. We show that $\mathcal{R}$ is a weak statebased simulation.
transition
Assume $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$. By Lemma 7.3.11 follows $[\![F(s')]\!]M'$. Consider the following cases for λ.

- $\lambda = \varepsilon$: By reflexivity of $\longrightarrow^*$ follows $\langle S''^k, M\rangle \xrightarrow{\langle\ \rangle}{}^* \langle S''^{k'}, M'\rangle$ with $k' = k$.
- $\lambda = \sigma$: From $[\![F(s)]\!]M$ follows $s \in \mathbb{S}_{\mathcal{L}(swap'')}$. Then, by Lemma 3.3.22 follows $\langle S'', M\rangle \xrightarrow{\sigma} \langle S''^{k''}, M'\rangle$ with $k'' \geq 1$. By (N2) and the definition of $\longrightarrow^*$ follows $\langle S''^k, M\rangle \xrightarrow{\sigma}{}^* \langle S''^{k'}, M'\rangle$ with $k' = k - 1 + k''$.

Hence $(\langle R'', M'\rangle, \langle S''^{k'}, M'\rangle) \in \mathcal{R}$.
termination
If $s \equiv \mathsf{skip}$ then $a_i = 0$ for all $i : 0 \leq i \leq N$. Hence by $[\![F(s)]\!]M$ follows $[\![\dagger swap'_i]\!]M$ for all $i : 0 \leq i \leq N$. Then $[\![\dagger swap'']\!]M$ and by a straightforward derivation $\langle S'', M\rangle \xrightarrow{\varepsilon}{}^* \langle \mathsf{skip}, M\rangle$. □

The Bubble Sort strategy we saw earlier executes $swap_i$'s in a fixed order; i.e. it is oblivious to its input. In contrast, the Ripple Sort schedule is adaptive; i.e. its behaviour depends on the values of the input. For instance, if the input is sorted, then the schedule terminates after $N - 1$ failing attempts at executing $swap'_i$ (for $i : 1 \leq i < N$) (thereby beating *Quick Sort* which takes $O(N^2)$ time for sorted input). Because Ripple Sort is adaptive, it will perform fewer swap's than Bubble Sort.

We proceed by showing that the Bubble Sort schedule derived in the previous section is a (weak) refinement of Ripple Sort.

Lemma 7.3.13 $BS(1, N) \precapprox_{M_0} R$

Proof We use the predicates Idx, Ord and Min' defined in Section 7.3.2. Let

$$\mathcal{R} = \{(\langle BS(n, m), M\rangle, \langle s, M\rangle) \mid Idx(n, m), [\![Ord(n)]\!]M, [\![Min'(n, m)]\!]M, [\![F(s)]\!]M\}$$

We show that $\mathcal{R}$ is a weak statebased simulation.
transition
Suppose $\langle BS(n, m), M\rangle \xrightarrow{\lambda} \langle t, M'\rangle$. This was derived, by (N9),

from $\langle swap_m; BS(n, m-1), M\rangle \xrightarrow{\lambda} \langle BS(n, m-1), M'\rangle$.
Analogous to Lemma 7.3.8 follows $t \equiv BS(n', m')$ such that $Idx(n', m')$, $[\![Ord(n')]\!]M'$ and $[\![Min'(n', m')]\!]M'$. Consider the following cases for λ.

- $\lambda = \varepsilon$: Then $M' = M$ and by reflexivity of $\longrightarrow^*$ follows $\langle s, M\rangle \xrightarrow{\langle\ \rangle}{}^* \langle s, M\rangle$. Hence $[\![F(R')]\!]M$ and $(\langle BS(n, m-1), M'\rangle, \langle R', M'\rangle) \in \mathcal{R}$.
- $\lambda = \sigma = \{(m-1, y), (m, x)\}/\{(m-1, x), (m, y)\}$ where $x > y$. From $[\![F(s)]\!]M$ follows $s \equiv \Pi_{i=0}^{N} R_i^{a_i}$ with $a_m \geq 1$. By (N1), follows $\langle R_{m-1}, M\rangle \xrightarrow{\sigma} \langle R_{m-2} \parallel R_m, M'\rangle$. Then, by (N2) and the definition of $\longrightarrow^*$, follows $\langle s, M\rangle \xrightarrow{\sigma}{}^* \langle s', M'\rangle$. By Lemma 7.3.11 follows $[\![F(s')]\!]M'$, hence $(\langle BS(n, m-1), M'\rangle, \langle s', M'\rangle) \in \mathcal{R}$.

termination
If $BS(n, m) \equiv \mathsf{skip}$, then $n = m \geq N - 1$. Then by $[\![Ord(n)]\!]M$ follows $\forall i : 0 \leq i \leq N : [\![\dagger swap'_i]\!]M$. Hence for all $i : 0 \leq i \leq N$ we derive, by (N0), $\langle swap'_i \to (R_{i-1} \parallel R_{i+1}), M\rangle \xrightarrow{\varepsilon} \langle \mathsf{skip}, M\rangle$. Then by (N3) and the definition of $\longrightarrow^*$ follows $\langle s, M\rangle \xrightarrow{\varepsilon}{}^* \langle \mathsf{skip}, M\rangle$.

Finally, it is straightforward to verify that $(\langle BS(1, N), M_0\rangle, \langle R, M_0\rangle) \in \mathcal{R}$. □

The Ripple Sort schedule provides a lot of opportunity for parallel execution. One way of exploiting this parallelism is by selecting sets of disjoint pairs of elements. For instance, either all $swap'_i$'s where i is odd or i is even can be executed in parallel. Alternatingly executing the $swap'_i$'s for even indices and odd indices yields a schedule called Odd-Even Transposition Sort (see [81],[102]). In the refinement ordering of sorting schedules, Odd-Even Transposition Sort should be situated between Ripple Sort and Bubble Sort.

7.3.4 Selection Sort

A family of sorting techniques is based on the idea of repeated selection: first find the smallest key and place it at the foremost position; then select the next smallest and so on. This idea is the basis of the coordination strategy that we derive in this section. The derivation proceeds from S' (7.15).

Outer Loop

In this section we will illustrate that the aforemention strategy of repeated selection arises naturally by decomposing the most general sorting schedule S' (7.15) with respect to the index-variable i of the constituent rewrite rule $swap'$.

By Lemma 7.3.1 follows that the variable i can only be matched to values from the interval $[1, N-1]$. For every value from this interval we introduce a strengthening $swap_k$ of $swap'$ and a corresponding schedule S_k. Define, for all $k : 1 \leq k < N$,

$$\begin{array}{lcl} swap_k & \widehat{=} & (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow 1 \leq i < j \leq N \wedge i = k \wedge x > y \\ S_k & \widehat{=} & !(swap_k \rightarrow S_k) \end{array} \tag{7.22}$$

Our repertoire of convex laws suggests that S' can be refined by the parallel or sequential composition of the schedules S_k. To decide which kind of composition is possible we investigate whether the schedules S_k interfere with each. To this end we study the stability of the termination properties established at termination of S_k.

At termination of S_k, the key at position k is smaller than or equal to the keys at positions greater than k. Formally, by Lemma 3.3.31 follows that NS_k, defined below, holds at termination of S_k.

$$NS_k \Leftrightarrow \forall i, j, x, y : (i,x),(j,y) \wedge i = k \wedge 1 \leq i < j \leq N : x \leq y \tag{7.23}$$

The predicate NS_k does not say anything about keys at positions smaller than k. In particular, there may be some key v_m at a position $m < k$ which is larger than v_k. Then S_m may perform a rewrite $swap(m,k)$ which could invalidate NS_k. Hence, there may be interference between the components S_i if they are executed in parallel. Consequently, S' may not be refined by the parallel composition $\Pi_{i=1}^{N-1} S_i$'s.

The above counter-example does not apply to $k = 1$ because, by Lemma 7.3.1, there are no elements with indices smaller than 1. Hence, once NS_1 is established, it cannot be invalidated by any of the other S_k's (for which $k \geq 2$); i.e. NS_1 is stable. Furthermore, if NS_1 holds, then NS_2 is stable. In general, NS_k is stable after the termination conditions of all preceding components S_i with $1 \leq i < k$ have been established. Hence the sequential composition of the components S_k seems a promising direction.

We proceed by verifying the preconditions of Lemma 6.2.11. The first precondition, $\natural swap' \Leftrightarrow (\exists i : \natural swap_i)$, follows from $(\exists k : 1 \leq k < N) \Leftrightarrow (\exists i : 1 \leq i < N : i = k)$.

The second precondition requires that the conjunction of the termination conditions of the components S_i for $i : 1 \leq i \leq k$ is stable. To prove this, we define AS_k (for all

$k : 1 \leq k < N)$

$$AS_k \quad \Leftrightarrow \quad (\forall i : 1 \leq i \leq k : NS_i)$$

or equivalently

$$AS_k \quad \Leftrightarrow \quad (\forall i : 1 \leq i \leq k : (\forall j, x, y : (i, x), (j, y) : i < j \leq N : x \leq y))$$

Lemma 7.3.14 $\forall k : 0 \leq k < N :$ *stable* AS_k

Proof Suppose $[\![AS_k]\!]M$ for some $k : 1 \leq k < N$ and let $M' = M[\sigma]$ where $\sigma = \{(i, y), (j, x)\}/\{(i, x), (j, y)\}$ with $i < j$ and $x > y$. From $[\![AS_k]\!]M$ follows $k < i < j \leq N$. All keys at positions at most k remain smaller than or equal the keys at positions greater than k by exchanging keys at positions greater than k. Hence $[\![AS_k]\!]M'$. □

The schedule $Select(1)$, defined below, executes the components S_k sequentially in increasing order of k.

$$Select(k) \quad \widehat{=} \quad k < N \triangleright (S_k; Select(k + 1)) \quad \text{for } k \geq 1 \tag{7.24}$$

By Lemma 6.2.11 follows $Select(1) \lesssim^{\diamond}_{M_0} S'$.

Remark 1: As alternative to the preceding derivation, we could also have taken a dual approach by splitting the domain of j. To this end, we define, for all $1 < k \leq N$

$$\begin{array}{lcl} swap'_k & = & (i, x), (j, y) \mapsto (i, y), (j, x) \Leftarrow 1 \leq i < j \leq N \wedge j = k \wedge x > y \\ S'_k & \widehat{=} & !(swap'_k \rightarrow S'_k) \end{array}$$

Here S'_N puts the largest element at position N. Clearly this property is stable. For this decomposition, the input can be sorted by starting with the maximum index position and moving towards successively smaller indices (up to 2). Such a strategy is described by the schedule $DualSelect(N)$ which is defined by

$$DualSelect(k) \quad \widehat{=} \quad k > 1 \triangleright (S_k; DualSelect(k - 1)) \quad \text{for } 1 < k \leq N$$

Remark 2: Heap Sort has in common with Selection sort that it operates according to a selection strategy: it consists of $N-1$ phases where during phase i, the i^{th} key is put in its proper position. In the process of selecting the i^{th} keys, the Heap Sort algorithm puts additional ordering on the remaining keys which facilitates the selection of the $i + 1^{th}$

key. The selection of the first key can be described using the following strengthening of $swap'$ from (7.15).

$$swap_h = (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow 1 \leq i < j \leq N \wedge (j = 2i \vee j = 2i+1)$$

Then the schedule $H \mathrel{\widehat{=}} !(swap_h \rightarrow H)$ puts the elements in heap structure (traditionally called the "*Heapify*" procedure). Implementations of Heap Sort then use a trick by which it is possible to maintain (and exploit) the additional ordering on keys in the interval $[2,\ldots,N]$ which remains to be sorted.

This trick involves the smart use of a data-structure, which can not be straightforwardly expressed in terms of strengthenings of *swap*. An interesting direction for future research would be to investigate whether the use of more structured data structures (than multisets) could help in defining such strategies.

Inner Loop

In this section we will refine the components S_k of the schedule *Select* which we derived in the previous section.

If AS_{k-1} holds, then S_k puts the minimum of the keys in the interval $[k,\ldots,N]$ at the k^{th} position. It establishes this by successively comparing the key at position k with all keys in the interval $[k+1,\ldots,N]$ and exchanging them if they are unordered. The order by which these comparisons should be executed is left unspecified.

To impose an order on these comparisons, we introduce the following collection of strengthenings. These are obtained by decomposing the domain of the index variable j of the rewrite rule $swap_k$ (7.22). We define one rewrite rule for every combination of k and l such that $1 \leq k < l \leq N$:

$$swap_{k,l} = (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow i = k \wedge j = l \wedge x > y \tag{7.25}$$

The rewrite rules $swap_{k,l}$ can be thought of as computing the minimum function "$min(x,y) =$ *if* $x \leq y$ *then* x *otherwise* y" for the elements (k,x) and (l,y). Because the *min* function is associative and commutative, it is tempting to think that any ordering of comparing the key at position k with the keys from positions $l \in [k+1,N]$ yields the required minimum. In fact, any arbitrary ordering would indeed be correct in a setting without interference. However, this is not the case if interference is allowed. We will briefly explain why this is so.

Computing the minimum of the keys in the interval $[k, \ldots, N]$ by performing the comparisons between the value at position k and the other positions in arbitrary order requires stability of the following property.

- Let $J \subseteq \{k+1, \ldots, N\}$ denote the set of indices whose keys have been compared with the key at position k.

$$\forall i, j, x, y : (i, x), (j, y) \wedge i = k \wedge j \in J : x \leq y \tag{7.26}$$

Consider elements (v, p) and (w, q) where $p, q : k < q < p \leq N$ within the unsorted interval $[k, N]$. Furthermore, let v be the unique minimum of the interval $[k, N]$ and let $p \notin J$ and $q \in J$; i.e. w has been compared with the key at position k, but the minimum v has not. From the fact that v is the minimum follows $v < w$. Hence a *swap* that is executed by the context (an interference) may exchange the keys v and w.

The minimum v that S_k is intended to find is now at position q which has already been visited ($q \in J$). Because the key currently at position k is not the minimum, it is larger than this minimum. Hence (7.26) does not hold (for $i = k$ and $j = q$).

Any schedule that compares the key at k once with all keys at higher positions, will not consider location q where the minimum is currently located again and will therefore fail to find it, and as a consequence fail to sort correctly. However, comparing the elements from high to low index-values (starting at N and successively decreasing down to $k+1$) does yield the correct result.

First, we will explain informally why this order of comparing keys works. Subsequently, we will present the formal derivation of the corresponding schedule.

Suppose that the keys in the positions $[1, \ldots, k]$ are in their proper (final) position. Let p be the lower bound of the interval $[p, N]$ of which the keys have been compared to the key at position $k+1$. Then, the minimum of the interval $[k+1, N]$ is located somewhere in the interval $[k+1, p]$. When an interfering *swap* occurs which moves the minimum, then

1. the minimum arrives at a lower index (because it is the smallest key), and

2. the minimum will not move below position $k+1$ because all keys below $k+1$ are smaller or equal to keys in the interval $[k+1, N]$.

Hence, the minimum is at some position within the interval $[k+1, p-1]$. By moving the upper bound p down to $k+1$, the minimum will ultimately be contained in the interval

$[k+1, k+1]$; i.e. be located at position $k+1$. The reason why interference cannot disturb this strategy is because the schedule and interference move the minimum in the same direction.

The schedule $GetMin(k, N)$, defined below, describes the strategy which performs the comparisons starting with position N and working successively down to k. Define, for all $k, l : 1 \leq k < l \leq N$,

$$\begin{array}{lcl} GetMin(k,l) & \widehat{=} & l > k \triangleright (T_{k,l}; GetMin(k, l-1)) \\ T_{k,l} & \widehat{=} & !(swap_{k,l} \rightarrow T_{k,l}) \end{array}$$

Next, we prove that, if $[\![AS_{k-1}]\!]M$ then $GetMin(k, N) \lesssim^{\diamond}_{M} S_k$. To this end, we verify the preconditions of Lemma 6.2.13.

Clearly $\natural swap_k \Rightarrow (\exists l : k < l \leq N : \natural swap_{k,l})$ for all $k : 1 \leq k < N$.

By Lemma 3.3.31 follows that the predicate $NT_{k,l}$, defined below, holds at termination of $T_{k,l}$.

$$NT_{k,l} = \forall i, j, x, y : (i, x), (j, y) \wedge i = k \wedge j = l : x \leq y$$

We introduce the predicate $AT_{k,l}$ to denote the conjunction of the termination predicates of $NT_{k,l}$ for all $l : k < l \leq N$. Define, for all $k, l : 1 \leq k < l \leq N$,

$$\begin{array}{lcl} AT_{k,l} & \Leftrightarrow & \forall i : l \leq i \leq N : NT_{k,i} \end{array}$$

or equivalently

$$\begin{array}{lcl} AT_{k,l} & \Leftrightarrow & \forall i, j, x, y : (i, x), (j, y) \wedge i = k \wedge l \leq j \leq N : x \leq y \end{array} \tag{7.27}$$

Now, we can show that if AS_k, then $\forall l : k < l \leq N :$ *stable* $AT_{k,l}$.

Lemma 7.3.15 *For all $k : 1 \leq k < N$: if AS_k, then $\forall l : k \leq l < N :$ stable $AT_{k,l}$.*

Proof Assume $[\![AS_k]\!]M$ for some $k : 1 \leq k < N$ and $[\![AT_{k,l}]\!]M$ for some $l : k < l \leq N$. Let $M' = M[\sigma]$ where $\sigma = \{(i, y), (j, x)\}/\{(i, x), (j, y)\}$ with $i < j$ and $x > y$. Consider the following cases for i and j:

- $i < j \leq k < l$: this contradicts AS_{k-1} which implies $x_i \leq y_j$.
- $k = i < j < l$: From $[\![AT_{k,l}]\!]M$ follows $[\![\forall m, z : (m, z) : l \leq m \leq N : x \leq z]\!]M$.
  From $i < j < l$ follows $\forall m : l \leq m \leq N \wedge (m, z) \in M \wedge (m, z') \in M' : z = z'$.
  From $y < x$ follows, by transitivity of $\leq$, $[\![\forall m, z' : (m, z') : l \leq m \leq N : y \leq z']\!]M'$.
  Hence $[\![AT_{k,l}]\!]M'$.

- $k = i < l \leq j \leq N$: this contradicts $AT_{k,l}$ which implies that $x_k \leq y_j$.

- $k < i < j < l \leq N$: Assume $(k, v) \in M$. From $k < i < j$ follows for $(k, v') \in M'$ that $v = v'$. From $[\![AT_{k,l}]\!]M$ follows $[\![\forall m, z : (m, z) : l \leq m \leq N : v \leq z]\!]M$.
  From $i < j < l$ follows $\forall m : l \leq m \leq N \wedge (m, z) \in M \wedge (m, z') \in M' : z = z'$.
  Hence $[\![AT_{k,l}]\!]M'$.

- $k < i < l \leq j \leq N$: Assume $(k, v) \in M$. From $k < i < j$ follows for $(k, v') \in M'$ that $v = v'$. From $[\![AT_{k,l}]\!]M$ follows $[\![\forall m, z : (m, z) : l \leq m \leq N : v \leq z]\!]M$.
  From $l \leq j \leq N$ and $y < x$ follows, by transitivity of $\leq$, that
  $\forall m : l \leq m \leq N \wedge (m, z) \in M \wedge (m, z') \in M' : z \leq z'$. By transitivity of $=$ and $\leq$ follows $[\![\forall m, z' : (m, z') : l \leq m \leq N : v' \leq z']\!]M'$. Hence $[\![AT_{k,l}]\!]M'$.

- $k < l < i < j \leq N$: Assume $(k, v) \in M$. From $k < i < j$ follows for $(k, v') \in M'$ that $v = v'$. From $[\![AT_{k,l}]\!]M$ follows $[\![\forall m, z : (m, z) : l \leq m \leq N : v \leq z]\!]M$.
  In particular, since $l < i < j \leq N$, we get $v \leq x$ and $v \leq y$. Hence, by transitivity of $=$ and $\leq$, follows $[\![\forall m, z' : (m, z') : l \leq m \leq N : v' \leq z']\!]M$.

□

By replacing every occurrence of S_k in $Select(1)$ by $GetMin(k, N)$ we obtain the schedule $Select'(1)$ which is defined by

$$Select'(k) \quad \widehat{=} \quad k < N \triangleright (GetMin(k, N); Select'(k+1)) \quad \text{for } k \geq 1 \tag{7.28}$$

Because, by Lemma 6.2.18 follows $\forall M : M \in \mathcal{O}^\diamond(S_1; \ldots; S_k, M_0) : [\![AS_k]\!]M$, and by Lemma 7.3.14 follows *stable* AS_k, we get by Corollary 6.2.16 that $Select'(1) \lesssim^\diamond_{M_0} Select(1)$.

The final refinement follows from the fact that in any multiset which satisfies $AT_{k,l+1}$, execution of $swap_{k,l}$ establishes $AT_{k,l}$. As a consequence, execution of $swap_{k,l}$ disables itself. By Lemma 6.2.24 follows that if $[\![AT_{k,l+1}]\!]M$, then $swap_{k,l} \lesssim^\diamond_M T_{k,l}$.

If $[\![AS_{k-1}]\!]M$, then by Lemma 6.2.18 follows $\forall M' : M' \in \mathcal{O}^\diamond(T_{k,N}; T_{k,N-1}; \ldots; T_{k,l}, M) : [\![AT_{k,l}]\!]M'$. By Lemma 7.3.15 follows that if $[\![AS_{k-1}]\!]M$, then *stable* $AT_{k,l}$. Then, Corollary 6.2.16 justifies the refinement $Select''(1) \lesssim^\diamond_{M_0} Select'(1)$, where $Select''(k)$ is defined by

$$\begin{array}{lcl} Select''(k) & \widehat{=} & k < N \triangleright (GetMin'(k, N); Select''(k+1)) \\ GetMin'(k, l) & \widehat{=} & l > k \triangleright (swap_{k,l}; GetMin'(k, l-1)) \end{array}$$

The sorting strategy derived is called Straight Selection Sort by Knuth [81].

Concluding Remarks

The Selection Sort schedule was derived using convex refinement laws. The first two refinements were obtained by decomposing the domain of the index variables of the rewrite rule $swap'$. The interference properties of the schedules obtained from this decomposition suggested the sequential coordination strategy of the refining schedules.

7.3.5 Quicksort

The sequential coordination strategy of Selection Sort was suggested by the interference properties of the components that were obtained by decomposing the domain of the index variables of *swap*. In this section we illustrate a decomposition that allows the resulting schedules to be executed in parallel.

A condition suggested by the refinement laws for decomposing a problem into parallel tasks, is that these tasks may not interfere with each other. For the sorting problem, the absence of interference can be obtained by partitioning the data to be sorted into one subset of keys that are greater than some pivot and one subset of keys that are smaller than this pivot. These subsets can be sorted independently and a solution to the original problem consists of putting the sorted sequence of smaller values in front of the sorted sequence of larger values.

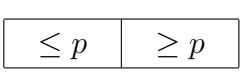
This decomposition yields two disjoint instances of the original problem that can be sorted according to the same strategy. Hence, this strategy can be applied recursively until subsets of size 1 are obtained.

In [72] Hoare first describes a program that sorts according to this strategy. Henceforth it has become known as *Quicksort*.

Divide-and-Conquer Structure

The core of the Quicksort algorithm is a partition procedure that rearranges the keys of the sequence to be sorted such that all keys at positions before a certain dividing line are less than the keys after this dividing line.

Let p be an arbitrary value from the domain of keys. The value p is referred to as the "*pivot*". Then the partitioning can be represented graphically as



We continue the derivation from S' (7.15). The strategy for refining S' consists of first decomposing S' into a schedule that consists of a partitioning-phase followed by a phase that performs the remaining work. Then, the schedule that performs the remaining work can be decomposed into two schedules that sort the partitions obtained by the preceding phase in parallel. Subsequently, we describe, in the next section, how the coordination structure of the partition-phase can be refined.

We start with defining a partition-schedule and a "remaining work" schedule. A rewrite rule for partitioning the keys can be constructed by strengthening the enabling condition of the rewrite rule $swap'$ such that it attempts to match x only with keys smaller than or equal to p and y only with keys greater than p. This yields the following strengthening, called $split_p$, of $swap'$.

$$split_p = (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow \left\{ \begin{array}{l} 1 \leq i < j \leq N \wedge x > y \\ x \geq p \wedge p \geq y \end{array} \right. \tag{7.29}$$

In order to use the convex refinement laws for decomposition, we need to obtain the complement of $split_p$ with respect to $swap'$. This complement takes care of the work that has to be done in addition to a partitioning of the data. This complement has to be a strengthening of $swap'$, say r, such that $\natural swap' \Rightarrow (\natural split_p \vee \natural r)$. Using propositional logic we can calculate that $swap_p$ (defined below) is a solution to this equation.

$$swap_p = (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow \left\{ \begin{array}{l} 1 \leq i < j \leq N \wedge x > y \\ x \leq p \vee p \leq y \end{array} \right. \tag{7.30}$$

We embed these rewrite rules in the schedules $Q \mathrel{\widehat{=}} !(split_p \rightarrow Q)$ and $R \mathrel{\widehat{=}} !(swap_p \rightarrow R)$.

Now that we have defined a partitioning schedule and a schedule for the remaining work, we set out to verify that their sequential composition is a refinement of the original schedule.

If Q terminates, it establishes T_p, defined by

$$T_p \quad \Leftrightarrow \quad \forall i,j,x,y : (i,x),(j,y) \wedge 1 \leq i < j \leq N : x \leq y \vee x < p \vee y > p \tag{7.31}$$

Hence Q rearranges the keys of the sequence with respect to the value p as required by the partition procedure. As a consequence from (7.31), there is at least one index k

which separates the keys $\leq p$ from those $\geq p$.

$$\begin{array}{l} \exists k, p : (k, p) : 1 \leq k \leq N : \\ \quad (\forall i, x : (i, x) : 1 \leq i \leq N : (i < k \Rightarrow x \leq p) \ \wedge \ (i \geq k \Rightarrow x \geq p)) \end{array} \tag{7.32}$$

In order to limit the selection of data to one of the parts that results from the partitioning of Q, the schedules that follow execution of Q should have access to the position of the pivot (e.g. the value of k). However, the schedule Q does not carry any parameter that represents this dividing line. This issue will be addressed in the next section.

Lemma 7.3.16 *stable* T_p

Proof Let $[\![T_p]\!]M$ and let $\sigma = \{(i, y), (j, x)\}/\{(i, x), (j, y)\}$ with $i < j$ and $x > y$ be a substitution of *swap*. We show that $[\![T_p]\!]M[\sigma]$ holds by considering the following cases

- $x < p$: by σ follows $y < p$. Hence, exchanging these keys maintains T_p.
- $x \geq p$: by σ follows $i < j$. Then by T_p follows $y \geq p$. Hence, exchanging these keys maintains T_p.

□

From stability of T_p follows that the property of Equation (7.32) is also stable.

Using the preceding results, Lemma 7.3.17 formally proves that the decomposition of S' into the sequential composition $Q; R$ is a refinement.

Lemma 7.3.17 $Q; R \lesssim^{\diamond} S'$

Proof Follows from Lemma 6.2.11, because

- $split_p, swap_p \lhd swap'$.
- Since $(x \geq p \wedge y \leq p) \vee (x \leq p \vee y \geq p)$, it follows that $[\![\natural swap']\!]M \Rightarrow ([\![\natural split_p]\!]M \vee [\![\natural swap_p]\!]M)$.
- By Lemma 7.3.16 follows *stable* T_p.
  From *stable* $\dagger swap'$ and $[\![\dagger swap']\!]M \Rightarrow ([\![\dagger split_p]\!]M \wedge [\![\dagger swap_p]\!]M)$ follows
  stable $(\dagger split_p \wedge \dagger swap_p)$.

□

Next, we show that the schedule R that performs the work that has to be done in addition to the partitioning, can be decomposed into two schedules R_1 and R_2 each of which sorts one subset obtained by the partitioning phase. We verify that these schedules do not interfere, hence may be executed in parallel.

Define R_1 and R_2 as most general schedules for rewrite rules $swap_1$ and $swap_2$ such that

$$\natural swap_p \Rightarrow (\natural swap_1 \vee \natural swap_2) \tag{7.33}$$

$$\begin{array}{l} R_1 \mathrel{\widehat{=}} !(swap_1 \to R_1) \text{ where } swap_1 = \\ (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow 1 \leq i < j \leq N \wedge x > y \wedge x \leq p \end{array} \tag{7.34}$$

$$\begin{array}{l} R_2 \mathrel{\widehat{=}} !(swap_2 \to R_2) \text{ where } swap_2 = \\ (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow 1 \leq i < j \leq N \wedge x > y \wedge y \geq p \end{array} \tag{7.35}$$

Termination of R_1 and R_2 establishes TR_1 and TR_2 respectively.

$$\begin{array}{rcl} TR_1 & = & \forall i,j,x,y : (i,x),(j,y) \wedge x \leq p : i < j \Rightarrow x \leq y \\ TR_2 & = & \forall i,j,x,y : (i,x),(j,y) \wedge y \geq p : i < j \Rightarrow x \leq y \end{array}$$

TR_1 states that all keys at most p are sorted. TR_2 states that all keys at least p are sorted. After termination of Q, both TR_1 and TR_2 are stable.

Lemma 7.3.18

1. *stable* $T_p \Rightarrow$ *stable* TR_1
2. *stable* $T_p \Rightarrow$ *stable* TR_2

Proof

1. Suppose $[\![T_p \wedge TR_1]\!]M$. Let $\sigma = \{(i,y),(j,x)\}/\{(i,x),(j,y)\}$ with $i < j$ and $x > y$ be a substitution of *swap* and let $M' = M[\sigma]$. Consider the following cases.
   - $y > p$: Then by σ follows $x > p$. These keys are outside the scope of TR_1 which therefore remains unaffected.
   - $y \leq p$: From σ follows $i < j$ and $x > y$. Then from $y \leq p$ and TP follows $x < p$. Then from TR_1 follows $x \leq y$. However, this contradicts $x > y$, hence this case cannot occur.

2. Analogous to the previous case.

□

Lemma 7.3.19 proves that R may be refined by the parallel composition of R_1 and R_2.

Lemma 7.3.19 *If* $[\![T_p]\!]M$, *then* $R_1 \parallel R_2 \lesssim^{\diamond}_M R$

Proof Follows from Lemma 6.2.7, because

- $swap_1, swap_2 \lhd swap_p$
- *stable* TR_1 and *stable* TR_2 follow from Lemma 7.3.18.
- $\natural swap_p \Rightarrow (\natural swap_1 \vee \natural swap_2)$ follows from (7.33).

□

Since Q establishes T_p and T_p is stable, Corollary 6.2.16 justifies that the refinement of Lemma 7.3.19 may be applied to the right of the sequential composition in $Q; R$. This yields $Q; (R_1 \parallel R_2) \lesssim^{\diamond} Q; R$.

Next, we strengthen the rewrites rules $swap_1$ and $swap_2$. These rules select elements with keys that are $x \leq p$ and $y \geq p$ respectively. The fact that Q has partioned the keys with respect to p, implies that we can indicate the range of the index-variables i and j where elements with values $\leq p$ and $\geq p$ are located. Assume as initial sequence $\overline{v} = \langle\, v_1, \ldots, v_N \,\rangle$ and let k be the position of the pivot p after termination of Q. Define

$$\begin{array}{l} R'_1 \mathrel{\widehat{=}} !(swap'_1 \rightarrow R'_1) \text{ where } swap'_1 = \\ (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow 1 \leq i < j \leq k \wedge x > y \end{array} \tag{7.36}$$

$$\begin{array}{l} R'_2 \mathrel{\widehat{=}} !(swap'_2 \rightarrow R'_2) \text{ where } swap'_2 = \\ (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow k \leq i < j \leq N \wedge x > y \end{array} \tag{7.37}$$

Lemma 7.3.20 *Let* $s \in \mathcal{L}(swap)$. *Then,*

1. *if* $[\![T_p]\!]M$ *then* $swap'_1 \rightarrow s =^{\diamond}_M swap_1 \rightarrow s$
2. *if* $[\![T_p]\!]M$ *then* $swap'_2 \rightarrow s =^{\diamond}_M swap_2 \rightarrow s$

Proof

1. The result follows from Lemma 6.2.1 because

   1. $[\![T_p]\!]M$ holds by assumption,
   2. *stable* T_p follows from Lemma 7.3.16,
   3. Clearly, $[\![T_p]\!]M \wedge [\![\natural swap_1]\!]M \Rightarrow [\![\natural swap'_1]\!]M$.

2. Analogous to the previous case.

□

Then, by precongruence of convex refinement, follows that if $[\![T_p]\!]M$, then $R'_1 =^{\diamond}_M R_1$ and $R'_2 =^{\diamond}_M R_2$. Hence if $[\![T_p]\!]$, then $R'_1 \parallel R'_2 =^{\diamond}_M R_1 \parallel R_2$. Then, because Q establishes T_p and T_p is stable, we get by Corollary 6.2.16, that $Q;(R'_1 \parallel R'_2) =^{\diamond}_{M_0} Q;(R_1 \parallel R_2)$.

By transitivity of refinement we get

$$Q;(R'_1 \parallel R'_2) \lesssim^{\diamond}_{M_0} S' \tag{7.38}$$

where schedules R'_1 and R'_2 are most general schedules for sorting, just as S', but limited to a specific interval of index-values. If we parameterize S' in the interval to be sorted, and parameterize Q in the interval to be partitioned, then we can rephrase (7.38) such that the recurrence of sorting as a subproblem becomes apparent.

Define $S_{l,h}$ as the most general schedule for sorting an interval $[l,h]$:

$$\begin{array}{l} S_{l,h} \mathrel{\widehat{=}} !(swap_{l,h} \to S_{l,h}) \text{ where } swap_{l,h} = \\ (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow l \le i < j \le h \wedge x > y \end{array} \tag{7.39}$$

Define $Split(l,h,p)$ as a schedule that partitions the interval $[l,h]$ with respect to some pivot p:

$$\begin{array}{l} Split(l,h,p) \mathrel{\widehat{=}} !(split_{l,h,p} \to Split(l,h,p)) \text{ where } split_{l,h,p} = \\ (i,x),(j,y) \mapsto (i,y),(j,x) \Leftarrow l \le i < j \le h \ \wedge \ x > y \ \wedge \ (x \ge p \wedge y \le p) \end{array} \tag{7.40}$$

Assume that k denotes the position of the pivot p after the partitioning phase (formally specified by (7.32)). Then, (7.38) can be rephrased as follows

$$Split(l,h,p);(S_{l,k} \parallel S_{k+1,h}) \lesssim^{\diamond}_{M_0} S_{l,h} \tag{7.41}$$

The value of k (in $S_{l,k}$ and $S_{k+1,h}$) depends on the value of the input and on the pivot chosen. Hence k cannot be known at this stage of the design, but can be determined

during execution. We should think of k as a meta-variable that is used in the derivation. In the next section we address the issue of eliminating this meta-variable from the schedule.

For any non-empty interval $[l, h]$, the schedules $S_{l,h}$ can be refined as in (7.41). Successive application of this refinement suggest that

$$Q(l, h) \lesssim^{\diamond} S_{l,h}$$

where

$$Q(l, h) \quad \widehat{=} \quad l < h \triangleright \mathit{Split}(l, h, p); (Q(l, k) \parallel Q(k + 1, h)) \tag{7.42}$$

The schedule $Q(l, h)$ describes the parallel divide-and-conquer structure of the Quicksort method.

We conclude this section with a refinement into a sequential version of Quicksort. This refinement follows straightforwardly using the law from Corollary 4.4.14.1.

$$QS(l, h) \lesssim^{\diamond} Q(l, h)$$

where

$$QS(l, h) \quad \widehat{=} \quad l < h \triangleright \mathit{Split}(l, h, p); (QS(l, k); QS(k + 1, h)) \tag{7.43}$$

In the next section we address the refinement of the subschedule *Split*.

Refinement of *Split*

In this section we concentrate on the *Split* schedule for partitioning an interval $[l, h]$. We set out to refine *Split* by a schedule that, as a result of partitioning an interval $[l, h]$, yields a value $k : l \leq k \leq h$ which separates the keys at most the pivot from those at least the pivot.

The basic idea for partitioning an interval is to successively inspect all key-values in the interval. Through inspection, we classify a key as at-most-pivot or at-least-pivot. All keys that are at-most–pivot are placed in an interval that starts at the lower bound and "grows" upwards. All keys that are at-least-pivot are placed in an interval that starts at the upper bound and "grows" downwards.

To be able to perform the partitioning "in-place" we inspect the key-values starting from the outer bounds and move inward as the intervals of classified keys grow. In this way, positions of keys that have been inspected coincide with positions for storing

classified keys.

At some stage during execution of this strategy, the lower- and upper interval meet. The position where this happens, indicates the position of the pivot.

The following strategy alternates between investigating the key at position i and investigating the key at position j. As usual, the parameters l and h indicate the lower- and upper bound of the interval that is to be partitioned. The parameter i indicates the upper bound of the interval of at-most-pivot keys and the parameter j indicates the lower bound of the interval of at-least-pivot keys. Hence, the interval $[l, i)$ contains keys that are at most p, the interval $(j, h]$ contains keys that are at least p and the interval $[i, j]$ is uninspected.

For the coordination strategy we will use the following rewrite rule

$$swap_{a,b,p} = (i, x), (j, y) \mapsto (i, y), (j, x) \Leftarrow \begin{cases} i < j \wedge i = a \wedge j = b \\ x > y \wedge x \geq p \wedge y \leq p \end{cases}$$

Based on these properties, we propose a strategy for partitioning that is described by two mutually recursive schedules *Rsplit* and *Lsplit*. The initial schedule is $RSplit(l, l, h, h)$ which takes the pivot p equal to the key that is initially at the lower bound position l

$$\begin{array}{lcl} Rsplit(l, i, j, h) & \widehat{=} & i < j \triangleright swap_{i,j,p} \rightarrow Lsplit(l, i + 1, j, h)[Rsplit(l, i, j - 1, h)] \\ Lsplit(l, i, j, h) & \widehat{=} & i < j \triangleright swap_{i,j,p} \rightarrow Rsplit(l, i, j - 1, h)[Lsplit(l, i + 1, j, h)] \end{array}$$

At termination of *Rsplit*, $i = j$ which indicates the position of the pivot (represented by k in the previous section). In order to forward this information to the subsequent schedule, we have to integrate the schedule that defines the recursive structure and the schedule that performs the partitioning into a single schedule definition. Define $QP(l, h) \widehat{=} Rsplit'(l, l, h, h)$ where

$$\begin{array}{lcll} Rsplit'(l, i, j, h) & \widehat{=} & i < j \triangleright & swap_{i,j,p} \rightarrow Lsplit'(l, i + 1, j, h)[Rsplit'(l, i, j - 1, h)] \\ & & & [QP(l, l, i, i) \parallel QP(i + 1, i + 1, h, h)] \\ Lsplit'(l, i, j, h) & \widehat{=} & i < j \triangleright & swap_{i,j,p} \rightarrow Rsplit'(l, i, j - 1, h)[Lsplit'(l, i + 1, j, h)] \\ & & & [QP(l, l, i, i) \parallel QP(i + 1, i + 1, h, h)] \end{array} \tag{7.44}$$

The congruent notions of refinement (i.e. stateless and convex) cannot be used to justify the modular use of the refinement $Rsplit(l, l, h, h) \precapprox Split(l, h, p)$ because these notions take interference into account that invalidates this refinement.

Instead, the refinement of $S_{l,h}$ by $QP(l,h)$ has to be proven using statebased simulation techniques. Because the techniques involved in statebased simulation proofs are illustrated elsewhere, we do not elaborate on them here.

7.3.6 Concluding Remarks

The following figure depicts the family tree of sorting algorithms ordered by the (weak statebased) refinement relation. The categorization follows the same classes as [60].

Figure 7.3: Lattice of Refinements of Sorting Schedules

There are interesting similarities between the derivation of sorting programs we presented and the one described in the context of logic programming by Darlington in [40].

Darlington works with a concrete data representation in terms of lists. Subsequent derivations are guided by decomposing the data structure; e.g. splitting this list in two: a head and a tail, or a first half and a second half.

These are essentially the same strategies as the ones that were used in our derivation of Selection Sort and Quicksort. However, in our case, these steps were constructed formally

through manipulation of the enabling condition of the rewrite rules of the schedule, hence by manipulation of the control structure rather than the data structure.

7.4 Single Source Shortest Paths

In this section we apply our development method to a graph problem that is known as the single source shortest paths problem. Pursuing separation between computation and coordination we shall first specify the basic computation that is required to solve the problem in Gamma. After that we shall relate several coordination strategies.

The problem description is as follows. Assume we are given a directed graph with vertices $V = \{1, \ldots, n\}$. A function L associates with every edge (u, v) a non-negative length $L(u, v)$. If there is no edge between vertices u and v, then we take $L(u, v) = \infty$. Moreover $L(u, u) = 0$ for all vertices u. Given a source vertex s, the problem is to determine for every vertex v, the length of a shortest path from s to v.

As data representation we use pairs (v, x), where v is a vertex number and x is the length of a path from the source s to v. Initially the length of a path from s to itself is set to 0; the lengths of the paths to other vertices are set to ∞. The initial multiset contains one element (v, x) for every vertex in the graph

$$M_0 = \{(s, 0)\} \cup \{\ (v, \infty)\ \mid\ 1 \leq v \leq n, v \neq s\}$$

A Gamma program for solving this problem is given by the rule:

$$\mathit{find} \mathrel{\widehat{=}} (u, x), (v, y) \mapsto (u, x), (v, x + L(u, v)) \Leftarrow x + L(u, v) < y$$

This program works by constructing shorter paths than the ones that are currently recorded. To this end, the rewrite rule *find* selects vertices u and v such that the length of the path from s to v via u is smaller than the length of the shortest path to v found so far. If such vertices can not be found, then there is no path of which the length can be decreased, hence all shortest paths have been found.

A formal correctness proof of this program (using a program logic similar to the one presented in Chapter 2) can be found in [79]. For future use we give two invariants of the program *find*: (1) every vertex of the graph is represented by exactly one element and (2) there are no elements other than the ones with indices in the range $[1, n]$.

Lemma 7.4.1

1. *invariant* $\forall i : 1 \leq i \leq n : (\#(v, x) : v = i) = 1$
2. *invariant* $\forall (v, x) : 1 \leq v \leq n$.

Proof Straightforward from the definition of *find*. □

7.4.1 A First Refinement

Though the program performs the required computation, it is hopelessly inefficient due to its unstructured search through the graph. We may coordinate the program's activities into a coherent searching strategy by adjoining the program with a coordination component. The starting point for the development of the coordination component is the most general schedule

$$S \mathrel{\widehat{=}} !(\mathit{find} \rightarrow S) \tag{7.45}$$

A more deterministic search strategy consists of conducting a directed search on the graph starting from the source. From a given vertex u the search proceeds by attempting to construct shorter paths to all adjacent vertices v (in no preferred order). If an attempt succeeds, the search continues from v; otherwise the search at v is aborted.

To define a schedule that expresses this strategy we first need a rewrite rule which allows us to indicate exactly which vertices should be selected next. To this end, we will use the strengthening $\mathit{find}_{u,v}$

$$\mathit{find}_{u,v} = (i,x),(j,y) \mapsto (i,x),(j,x+L(i,j)) \Leftarrow x+L(i,j) < y \wedge i = u \wedge j = v$$

By convention we abbreviate this to

$$\mathit{find}_{u,v} = (u,x),(v,y) \mapsto (u,x),(v,x+L(u,v)) \Leftarrow x+L(u,v) < y$$

Now, we specify our strategy by $\mathit{Search}(s)$, where

$$\mathit{Search}(u) \quad \mathrel{\widehat{=}} \quad (\Pi_{v=1}^{n} \mathit{find}_{u,v} \rightarrow \mathit{Search}(v))$$

Note that this schedule essentially describes a domain decomposition of the vertex variable j. However, because the searches for all values of j are interrelated (they might at some stage visit the same vertex), the encapsulating schedule could not be decomposed (analogous to decompositions in earlier cases).

To start off the the illustration of the derivation method, we shall prove that *Search* is a correct refinement of the most general schedule S. Because schedule *Search* still exhibits highly nondeterministic behaviour (it traverses the paths in the graph in any (possibly in parallel) order), the refinement techniques from Chapter 4 enable us to further transform the schedule *Search* into more deterministic versions.

We define a schedule GS which describes all forms that the schedule *Search* may

evolve into during execution. Define, for any multiset F over $V \times V$,

$$GS(F) \mathrel{\widehat{=}} (\Pi_{(u_1,u_2)\in F} \mathit{find}(u_1, u_2) \rightarrow \mathit{Search}(u_2))$$

The schedule GS enjoys the following two properties:

1. The schedule *Search* can be equivalently expressed as

$$\mathit{Search}(u) \equiv GS(U) \text{ where } U = \{(u, v) \mid v \in V\} \tag{7.46}$$

2. Commutativity of parallel composition implies the following identity

$$GS(F \cup F') \equiv GS(F) \parallel GS(F') \tag{7.47}$$

   Note that $\cup$ is used to denote multiset union which is defined, together with multiset subtraction (denoted $\ominus$), in Appendix A.2.

We introduce some predicates that will be used in Lemma 7.4.4. Predicate *UI* (for "unique index") repeats the invariant from Lemma 7.4.1. Predicate *TD* (for "to do") states that if for some pair of vertices there is no rewrite in the schedule GS, then there is no shorter path along the edge formed by these vertices. Hence, the multiset F is used to keep track of the edges which might potentially lead to shortest paths. Define

$$\mathit{UI} \quad \Leftrightarrow \quad \forall i : 1 \leq i \leq n : (\#(v, x) : v = i) = 1$$

$$\mathit{TD}(F) \quad \Leftrightarrow \quad \forall v_1, v_2, x_1, x_2 : (v_1, x_1), (v_2, x_2) : (v_1, v_2) \notin F : x_2 \leq x_1 + L(v_1, v_2)$$

In the next two lemmas we consider the invariance of the above properties for the family of schedule GS. These lemmas focus on single-step transitions such that every transition corresponds to the execution of a single rewrite rule $\mathit{find}_{u,v}$. First we consider ε-transitions (which denote failing execution).

Lemma 7.4.2 *Let $\langle GS(F), M\rangle$ be a configuration such that $[\![UI]\!]M$ and $[\![TD(F)]\!]M$. If $\langle GS(F), M\rangle \xrightarrow{\varepsilon} \langle s', M'\rangle$ is a single-step transition, then*

1. *$s' \simeq GS(F')$ where $F' = F \ominus \{(u, v)\}$ for some $(u, v) \in F$*

2. *$[\![UI]\!]M'$*

3. $[\![TD(F')]\!]M'$

Proof

1. By $(N0)$ and $(N2)$ we derive $s' \equiv GS(F')$ where $F' = F \ominus \{(u,v)\}$.
   By Lemma 4.4.9 follows $s' \simeq GS(F')$.

2. Because $M' = M$, clearly $[\![UI]\!]M'$.

3. By $[\![UI]\!]M$ follows $(u,x),(v,y) \in M$. From (N0) follows $[\![\dagger \mathit{find}_{u,v}]\!]M$, hence $x + L(u,v) \geq y$. Consequently, from $[\![TD(F)]\!]M$ follows

   $$\forall v_1, v_2, x_1, x_2 : (v_1,x_1),(v_2,x_2) : (v_1,v_2) \notin (F \ominus \{(u,v)\}) : x_2 \leq x_1 + L(v_1,v_2)$$

   From $F' = F \ominus \{(u,v)\}$ and $M' = M$ follows $[\![TD(F')]\!]M'$.

□

Next, we show that the properties are preserved by σ-transitions.

Lemma 7.4.3 *Let $\langle GS(F), M\rangle$ be a configuration such that $[\![UI]\!]M$ and $[\![TD(F)]\!]M$. If $\langle GS(F), M\rangle \xrightarrow{\sigma} \langle s', M'\rangle$ is a single-step transition, then*

1. $s' \simeq GS(F')$ *where* $F' = F \ominus \{(u,v)\} \cup \{(v,w) \mid w \in V\}$

2. $[\![UI]\!]M'$

3. $[\![TD(F')]\!]M'$

Proof Because σ is the label of a successful single-step transition from M we have $\sigma = \{(v,y')\}/\{(v,y)\}$ where $y' = x + L(u,v)$ and $y' < y$ for some $(u,x),(v,y) \in M$ such that $(u,v) \in F$.

1. By $(N1)$ and $(N2)$ follows $s' \equiv GS(F \ominus \{(u,v)\}) \parallel \mathit{Search}(v)$. By (7.46) follows $s' \equiv GS(F \ominus \{(u,v)\}) \parallel GS(F')$ where $F' = \{(v,w) \mid w \in V\}$. By (7.47) follows $s' \equiv GS(F'')$ where $F'' = F \ominus \{(u,v)\} \cup \{(v,w) \mid w \in V\}$.

2. Follows from $\mathit{find}_{u,v} \lhd \mathit{find}$ and the invariant from Lemma 7.4.1.1.

3. We start by deriving some auxiliary results.

(1) As a special case of $[\![TD(F)]\!]M$ we have

$$\begin{array}{l}\forall v_1, v_2, x_1, x_2 : (v_1, x_1), (v_2, x_2) \in M : \\ (v_1, v_2) \notin F \wedge (v_2, x_2) = (v, y) : y \leq x_1 + L(v_1, v)\end{array}$$

By $[\![UI]\!]M'$, the $M' = M[\sigma]$ and $y' < y$ follows

$$\begin{array}{l}\forall v_1, v_2, x_1, x_2 : (v_1, x_1), (v_2, x_2) \in M' : \\ (v_1, v_2) \notin F \wedge (v_2, x_2) = (v, y') : y' \leq x_1 + L(v_1, v)\end{array}$$

(2) By $[\![UI]\!]M'$ and $M' = M[\sigma]$ follows

$$\begin{array}{l}\forall v_1, v_2, x_1, x_2 : (v_1, x_1), (v_2, x_2) \in M' : \\ v_1 = u \wedge v_2 = v : x_2 = x_1 + L(v_1, v_2)\end{array}$$

Using these results, we reason as follows. From $M' = M[\sigma]$ and $[\![TD(F)]\!]M$ follows

$$\begin{array}{ll} & \forall v_1, v_2, x_1, x_2 : (v_1, x_1), (v_2, x_2) \in M' : \\ & (v_1, v_2) \notin F \wedge v_1 \neq v \wedge v_2 \neq v : x_2 \leq x_1 + L(v_1, v_2) \\ \Rightarrow & \qquad \{ \text{ by (1) } \} \\ & \forall v_1, v_2, x_1, x_2 : (v_1, x_1), (v_2, x_2) \in M' : \\ & (v_1, v_2) \notin F \wedge v_1 \neq v : x_2 \leq x_1 + L(v_1, v_2) \\ \Rightarrow & \qquad \{ \text{ set calculus } \} \\ & \forall v_1, v_2, x_1, x_2 : (v_1, x_1), (v_2, x_2) \in M' : \\ & (v_1, v_2) \notin (F \cup \{(v, w) \mid w \in V\}) : x_2 \leq x_1 + L(v_1, v_2) \\ \Rightarrow & \qquad \{ F' \cup \{(u, v)\} = F \cup \{(v, w) \mid w \in V\} \} \\ & \forall v_1, v_2, x_1, x_2 : (v_1, x_1), (v_2, x_2) \in M' : \\ & (v_1, v_2) \notin (F' \cup \{(u, v)\}) : x_2 \leq x_1 + L(v_1, v_2) \\ \Rightarrow & \qquad \{ \text{ by (2) and } [\![UI]\!]M' \} \\ & \forall v_1, v_2, x_1, x_2 : (v_1, x_1), (v_2, x_2) \in M' : \\ & (v_1, v_2) \notin F' : x_2 \leq x_1 + L(v_1, v_2)\end{array}$$

Which proves $[\![TD(F')]\!]M'$. □

Now that we have set up some useful lemmas, we proceed by proving the main refinement.

Lemma 7.4.4 *Let* $S \mathrel{\widehat{=}} !(\mathit{find} \to S)$. *Then* $\langle \mathit{Search}(s), M_0\rangle \precapprox \langle S, M_0\rangle$

Proof Let

$$\mathcal{R} = \{(\langle GS(F), M\rangle, \langle S^k, M\rangle) \mid [\![UI]\!]M \wedge [\![TD(F)]\!]M, k \geq 1\}$$

We show that $\mathcal{R}$ is a weak statebased simulation up-to weak statebased refinement. To this end, consider the following cases.
transition
If $\langle GS(F), M\rangle \xrightarrow{\lambda} \langle s', M'\rangle$, then by Lemma 3.2.5 there exist $\lambda_1, \ldots, \lambda_n, n \geq 1$ such that

$$\langle s_0, M_0\rangle \xrightarrow{\lambda_1} \langle s_1, M_1\rangle \ldots \xrightarrow{\lambda_i} \ldots \langle s_{n-1}, M_{n-1}\rangle \xrightarrow{\lambda_n} \langle s_n, M_n\rangle$$

where $\langle GS(F), M\rangle = \langle s_0, M_0\rangle$, $\langle s_n, M_n\rangle = \langle s', M'\rangle$ and each transition is single-step. By induction on the length of the transition sequence it follows by Lemma 7.4.2 and Lemma 7.4.3 that $s' \simeq GS(F')$ such that $[\![UI]\!]M'$ and $[\![TD(F')]\!]M'$. From $\mathcal{L}(GS(F)) \lhd \mathit{find}$ follows, by Lemma 3.3.22, that $\langle S, M\rangle \xrightarrow{\lambda} \langle S^{k'}, M'\rangle$ for some $k' \geq 1$. By $(N2)$ we derive $\langle S^k, M\rangle \xrightarrow{\lambda} \langle S^{k-1+k'}, M'\rangle$ with $k - 1 + k' \geq 1$.
Hence $\langle s', M'\rangle \precsim \mathcal{R} \precsim \langle S^{k-1+k'}, M'\rangle$.
termination
$GS(F) \equiv \mathsf{skip}$ only if $F = \emptyset$. By $[\![TD(\emptyset)]\!]M$ follows $[\![\dagger \mathit{find}_{u,v}]\!]M$ for all $u, v \in V$. Then by a straightforward derivation follows $\langle S^k, M\rangle \xrightarrow{\varepsilon}{}^* \langle \mathsf{skip}, M\rangle$.

Next, we show that $(\langle GS(F_0), M_0\rangle, \langle S, M_0\rangle) \in \mathcal{R}$ where $F_0 = \{(s, v) \mid v \in V\}$. Clearly the schedules have the proper form and $[\![UI]\!]M_0$ holds. To verify $[\![TD(F_0)]\!]M_0$ we need to check that $\forall v_1, v_2, x_1, x_2 : (v_1, x_1), (v_2, x_2) \in M_0 : v_2 \notin \{s\} : x_2 \leq x_1 + L(v_1, v_2)$. This follows from the initialization (v_i, ∞) of all vertices $v_i \in V - \{s\}$.

From (7.46) and Lemma 5.6.1 follows $GS(F_0) \approx_{M_0} \mathit{Search}(s)$.
Hence $\langle \mathit{Search}(s), M_0\rangle \precapprox \langle S, M_0\rangle$ □

A large part of proving statebased refinements consists of showing that the invariants are preserved by all possible transitions of the schedule. If a schedule consists of the parallel composition of k components, then the multi-step semantics may give rise to an exponential number, $2^k - 1$, of possible transitions.

However, using Lemma 3.2.5 it suffices to show that the property is preserved by every possible sequence of single-step transitions (with identical effect on the multiset).

This can be proven by showing that the property is preserved by every individual single-step transition within this sequence. This effectively reduces reasoning about the parallel behaviour of schedules to reasoning about their interleaved behaviour.

The schedule *Search* will be our point of departure for further refinement. In the following sections, we will derive, by algebraic reasoning, schedules that impose depth-first and (parallel) breadth-first orderings on the execution of the shortest paths program.

For ease of manipulation, we write the schedule $Search(s)$ in a different, but equivalent form, as

$$\begin{array}{lcl} Search(u) & \widehat{=} & Visit(1,u) \\ Visit(i,u) & \widehat{=} & (i \leq n) \triangleright (\mathit{find}(u,i) \rightarrow Search(i)) \parallel Visit(i+1,u) \end{array}$$

7.4.2 Depth-First Search

The schedule *Search* visits neighbouring vertices in arbitrary order. Replacing this arbitrary order by strictly sequential ordering yields a strategy where the vertices are visited in a depth-first order. Such a strategy is described by the schedule $DepthFirst(s)$ where

$$\begin{array}{lcl} DepthFirst(u) & \widehat{=} & DFVisit(1,u) \\ DFVisit(i,u) & \widehat{=} & (i \leq n) \triangleright (\mathit{find}(u,i) \rightarrow DepthFirst(i)); DFVisit(i+1,u) \end{array}$$

Next, we show that the depth-first ordering is a correct refinement of the parallel strategy.

Lemma 7.4.5 $DepthFirst(s) \leqslant Search(s)$

Proof Immediately using Corollary 4.4.14.1. □

7.4.3 Breadth-First Schedule

An alternative to Corollary 4.4.14.1 for introducing sequential behaviour is presented by Lemma 4.4.10.1. It appears that repetitive application of this law to the parallel composition of schedule *Search* ultimately yields a breadth-first ordering.

As is standard, the schedule for a breadth-first search maintains a sequence of vertices that are yet to be visited. We write $v \cdot \overline{w}$ to denote the sequence $\overline{w}$ with the element v prepended, and $\overline{w} \cdot u$ for u appended to $\overline{w}$. We use $\langle\ \rangle$ to denote the empty sequence.

A breadth-first ordering can now be expressed by the following recursive schedule definitions:

$$\begin{array}{lcl} \mathit{BreadthFirst}(\langle\,\rangle) & \widehat{=} & \mathsf{skip} \\ \mathit{BreadthFirst}(u \cdot \overline{w}) & \widehat{=} & \mathit{BFVisit}(1, u, \overline{w}) \end{array}$$

where

$$\begin{array}{lcl} \mathit{BFVisit}(i, u, \overline{w}) & \widehat{=} & (i \leq n) \;\triangleright\; \mathit{find}(u, i) \rightarrow \mathit{BFVisit}(i+1, u, \overline{w} \cdot i) \\ & & \qquad\qquad\qquad\qquad\quad [\mathit{BFVisit}(i+1, u, \overline{w})] \\ & & \qquad\qquad [\mathit{BreadthFirst}(\overline{w})] \end{array}$$

The proof that breadth-first search is a correct refinement of the parallel strategy, i.e. $\mathit{BreadthFirst}(\langle\, s \,\rangle) \leqslant \mathit{Search}(s)$, is somewhat more involved than the case of a depth-first ordering. Therefore we present it here as a more detailed example of application of the refinement laws.

The proof is largely based on an intermediate result that we present first. We use $\overline{x} \# \overline{y}$ to denote the set of interleavings of sequences $\overline{x}$ and $\overline{y}$. More formally:

Definition 7.4.6 *Let $\overline{x}$ and $\overline{y}$ be two finite sequences, the set of their interleavings, denoted $\overline{x} \# \overline{y}$, is defined by*

$$\begin{array}{lcll} 1. & \langle\,\rangle \# \overline{x} & = & \{\overline{x}\} \\ 2. & \overline{x} \# \langle\,\rangle & = & \{\overline{x}\} \\ 3. & x_1 \cdot \overline{x'} \# y_1 \cdot \overline{y'} & = & \{x_1 \cdot \overline{z} \mid \overline{z} \in \overline{x'} \# (y_1 \cdot \overline{y'})\} \cup \{y_1 \cdot \overline{z} \mid \overline{z} \in (x_1 \cdot \overline{x'}) \# \overline{y'}\} \end{array}$$

Associativity of concatenation carries over onto interleaving. From the symmetry of interleaving, we deduce that # is a commutative operator; furthermore # has unit $\langle\,\rangle$. From Definition 7.4.6 follows that the interleaving operator "#" has the following properties:

$(I0)$ 1. $\overline{x} \# (\overline{y} \# \overline{z}) = (\overline{x} \# \overline{y}) \# \overline{z}$ associativity
2. $\overline{x} \# \overline{y} = \overline{y} \# \overline{x}$ commutativity

$(I1)$ If $\overline{w} \in \overline{w_1} \# \overline{w_2}$ and $\overline{w} = \langle\,\rangle$, then $\overline{w_1} = \langle\,\rangle$ and $\overline{w_2} = \langle\,\rangle$.

$(I2)$ If $u \cdot \overline{w} \in \overline{w_1} \# \overline{w_2}$, then $\overline{w_1} = u \cdot \overline{w_1'}$ and $\overline{w} \in \overline{w_1'} \# \overline{w_2}$ or $\overline{w_2} = u \cdot \overline{w_2'}$ and $\overline{w} \in \overline{w_1} \# \overline{w_2'}$.

$(I3)$ If $\overline{w} \in \overline{w_1} \# \overline{w_2}$, then $(\overline{w} \cdot i) \in (\overline{w_1} \cdot i) \# \overline{w_2}$.

Lemma 7.4.7 *Let* $\overline{w}, \overline{w_1}, \overline{w_2} \in V^*$ *be sequences such that* $\overline{w} \in \overline{w_1}\#\overline{w_2}$. *Let* $n \geq 0$, $1 \leq i \leq n$, *and* $u \in V$. *Then*

$BFVisit(i, u, \overline{w}) \leqslant BFVisit(i, u, \overline{w_1}) \parallel BreadthFirst(\overline{w_2})$

Proof

The result follows by showing that $\mathcal{R}$, defined as

$$\begin{aligned}\mathcal{R} \;=\; & \{(BFVisit(i, u, \overline{w}), BFVisit(i, u, \overline{w_1}) \parallel BreadthFirst(\overline{w_2})) \mid \\ & i \geq 0,\ 1 \leq u \leq n,\ w \in \overline{w_1}\#\overline{w_2},\ \overline{w_1}, \overline{w_2} \in V^*\ \}\end{aligned}$$

is a strong stateless simulation.

transition

We consider the possible transitions of $BFVisit(i, u, \overline{w})$. First consider the case $i \leq n$. There are two possible transitions:

1. Assume, $\langle find(u, i) \rightarrow BFVisit(i+1, u, \overline{w} \cdot i)[BFVisit(i+1, u, \overline{w})], M\rangle \xrightarrow{\sigma} \langle BFVisit(i+1, u, \overline{w} \cdot i), M'\rangle$

   Using (N8) and (E9) we get

   $$\langle BFVisit(i, u, \overline{w}), M\rangle \xrightarrow{\sigma} \langle BFVisit(i+1, u, \overline{w} \cdot i), M'\rangle$$

   Analogously, we get for $BFVisit(i, u, \overline{w_1})$

   $$\langle BFVisit(i, u, \overline{w_1}), M\rangle \xrightarrow{\sigma} \langle BFVisit(i+1, u, \overline{w_1} \cdot i), M'\rangle$$

   hence, by (N2), for $BFVisit(i, u, \overline{w_1} \cdot i) \parallel BreadthFirst(\overline{w_2})$:

   $$\langle BFVisit(i, u, \overline{w_1}) \parallel BreadthFirst(\overline{w_2}), M\rangle \xrightarrow{\sigma} \langle BFVisit(i+1, u, \overline{w_1} \cdot i) \parallel BreadthFirst(\overline{w_2}), M'\rangle$$

   From $\overline{w} \in \overline{w_1}\#\overline{w_2}$ follows by $(I3)$ that $\overline{w} \cdot i \in (\overline{w_1} \cdot i)\#\overline{w_2}$, hence
   $(BFVisit(i+1, u, \overline{w} \cdot i), BFVisit(i+1, u, \overline{w_1} \cdot i) \parallel BreadthFirst(\overline{w_2})) \in \mathcal{R}$.

2. The proof for $\langle find(u, i) \rightarrow BFVisit(i+1, u, \overline{w} \cdot i)[BFVisit(i+1, u, \overline{w})], M\rangle \xrightarrow{\varepsilon} \langle BFVisit(i+1, u, \overline{w}), M'\rangle$

is analogous to the previous case.

Next, we consider the case $i > n$ and $\overline{w} = u' \cdot \overline{w'}$. From $i > n$ follows $BFVisit(i, u, \overline{w}) \simeq BreadthFirst(\overline{w})$. By definition of *BreadthFirst*, (E9) and Lemma 4.4.9 follows $BreadthFirst(\overline{w}) \simeq BFVisit(1, u', \overline{w'})$. From $u' \cdot \overline{w'} \in \overline{w_1} \# \overline{w_2}$ follows by (I2) that $\overline{w_1} = u' \cdot \overline{w_1'}$ or $\overline{w_2} = u' \cdot \overline{w_2'}$. We consider these cases in turn.

- $\overline{w_1} = u' \cdot \overline{w_1'}$: Then

$$\begin{array}{ll} & BFVisit(i, u, \overline{w_1}) \parallel BreadthFirst(\overline{w_2}) \\ \simeq & \\ & BreadthFirst(\overline{w_1}) \parallel BreadthFirst(\overline{w_2}) \\ \simeq & \\ & BFVisit(1, u', \overline{w_1'}) \parallel BreadthFirst(\overline{w_2}) \end{array}$$

  The previous case ($i \leq n$), then gives
  $(BFVisit(1, u', \overline{w'}), BFVisit(1, u', \overline{w_1'}) \parallel BreadthFirst(\overline{w_2})) \in \mathcal{R}$.

- $\overline{w_2} = u' \cdot \overline{w_2'}$: Then

$$\begin{array}{ll} & BFVisit(i, u, \overline{w_1}) \parallel BreadthFirst(\overline{w_2}) \\ \simeq & \\ & BreadthFirst(\overline{w_1}) \parallel BreadthFirst(\overline{w_2}) \\ \simeq & \\ & BreadthFirst(\overline{w_1}) \parallel BFVisit(1, u_1', \overline{w_2'}) \end{array}$$

  The remainder of the proof is analogous to the previous case.

termination
$BFVisit(i, u, \overline{w}) \equiv \mathsf{skip}$ only if $i > n$ and $\overline{w} = \langle\,\rangle$. Then, from the definition of *BreadthFirst* follows: $BreadthFirst(\overline{w}) \simeq \mathsf{skip}$. From (I1) follows $\overline{w_1} = \langle\,\rangle$ and $\overline{w_2} = \langle\,\rangle$, hence by definition of *BreadthFirst* follows $BreadthFirst(\overline{w_1}) \parallel BreadthFirst(\overline{w_2}) \equiv \mathsf{skip}$. □

The essence of Lemma 7.4.7 is more clearly shown by the following corollary: the refining schedules (left hand side) visit the vertices in a deterministic order, while for the schedules on the right hand side this order is determined dynamically by the way parallel components interleave.

Corollary 7.4.8

1. *Let* $\overline{w}, \overline{w_1}, \overline{w_2} \in V^*$ *be sequences such that* $\overline{w} \in \overline{w_1} \# \overline{w_2}$.
   Then $BreadthFirst(\overline{w}) \leqslant BreadthFirst(\overline{w_1}) \parallel BreadthFirst(\overline{w_2})$.

2. *If* $\overline{w} = \langle\, w_1, \ldots, w_n \,\rangle$, *then* $BreadthFirst(\overline{w}) \leqslant \Pi_{i=1}^{n} BreadthFirst(\langle\, w_i \,\rangle)$.

Proof

1. Straightforward by the definition of *BreadthFirst* and Lemma 7.4.7.

2. From case 1 using induction on n.

□

We use the refinement from Lemma 7.4.7 as an algebraic law in the following calculation.

$$
\begin{array}{ll}
 & BFVisit(i, u, \overline{w}) \\
\simeq & \\
 & (i \leq n) \;\triangleright\; \mathit{find}(u, i) \rightarrow \quad BFVisit(i+1, u, \overline{w} \cdot i) \\
 & \qquad\qquad\qquad\qquad\quad [BFVisit(i+1, u, \overline{w})] \\
 & \qquad\quad [BreadthFirst(\overline{w})] \\
\leqslant & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{Lemma 7.4.7} \\
 & (i \leq n) \;\triangleright\; \mathit{find}(u, i) \rightarrow \quad BFVisit(i+1, u, \overline{w}) \parallel BreadthFirst(\langle\, i \,\rangle) \\
 & \qquad\qquad\qquad\qquad\quad [BFVisit(i+1, u, \overline{w})] \\
 & \qquad\quad [BreadthFirst(\overline{w})] \\
\leqslant & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{Lemmas 4.4.12, and 4.4.10.1} \\
 & (i \leq n) \;\triangleright\; (\mathit{find}(u, i) \rightarrow \quad BreadthFirst(\langle\, i \,\rangle)) \parallel BFVisit(i+1, u, \overline{w}) \\
 & \qquad\quad [BreadthFirst(\overline{w})]
\end{array}
$$

Finally, we prove the specific case $BreadthFirst(\langle\, s \,\rangle) \leqslant Search(s)$ as follows

$$
\begin{array}{ll}
 & BFVisit(i, u, \langle\,\rangle) \\
\simeq & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad def.\ BFVisit \\
 & (i \leq n) \;\triangleright\; (\mathit{find}(s, i) \rightarrow \quad BreadthFirst(\langle\, i \,\rangle)) \parallel BFVisit(i+1, s, \langle\,\rangle) \\
 & \qquad\quad [BreadthFirst(\langle\,\rangle)] \\
\simeq & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad def.\ BreadthFirst, \text{ Lemma 4.4.12} \\
 & (i \leq n) \;\triangleright\; (\mathit{find}(s, i) \rightarrow \quad BreadthFirst(\langle\, i \,\rangle)) \parallel BFVisit(i+1, s, \langle\,\rangle)
\end{array}
$$

The latter schedule term can be seen to equal $Visit(1, s)$, by substituting $Search(i)$ for $BreadthFirst(\langle\, i \,\rangle)$ and $Visit(i, u)$ for $BFVisit(i, u, \langle\, \rangle)$. Hence the refinement $BreadthFirst(\langle\, s \,\rangle) \leqslant Search(s)$ follows from the schedule definitions of *BreadthFirst* and *Search* and Lemma 5.3.15.

7.4.4 Parallel Breadth-first Search

We conclude the examples with a parallel version of breadth-first search that recursively divides the searching process into two if the amount of work exceeds some predefined threshold $k \geq 1$. A schedule that conducts this kind of search is a variation of the previous schedule and is defined as follows, where the schedule *ParBFVisit* is a renamed version of *BFVisit* from Section 7.4.3

$$
\begin{array}{lcl}
ParBF(\langle\, \rangle) & \widehat{=} & \mathsf{skip} \\
ParBF(\langle\, v_1, \ldots, v_m \,\rangle) & \widehat{=} & \\
\qquad (m > k) \;\triangleright\; & & ParBF(\langle\, v_1, \ldots, v_{m \,\mathtt{div}\, 2} \,\rangle) \parallel ParBF(\langle\, v_{m \,\mathtt{div}\, 2+1}, \ldots, v_m \,\rangle) \\
& & [ParBFVisit(1, v_1, \langle\, v_2, \ldots, v_m \,\rangle)] \\
ParBFVisit(i, u, \overline{w}) & \widehat{=} & (i \leq n) \;\triangleright\; find(u, i) \rightarrow ParBFVisit(i+1, u, \overline{w} \cdot i) \\
& & \qquad\qquad [ParBFVisit(i+1, u, \overline{w})] \\
& & [ParBF(\overline{w})]
\end{array}
$$

Lemma 7.4.9 $ParBF(\langle\, s \,\rangle) \leqslant Search(s)$

Proof
Let $\mathcal{R} = \{(\Pi_{j=1}^{m} ParBFVisit(i_j, u_j, \overline{w_j}), \Pi_{j=1}^{m}(\, Visit(i_j, u_j) \parallel \Pi_{k=1}^{|w_j|} Search(w_{j_k}))) \mid m \geq 0\}$.
The result follows by showing that $\mathcal{R}$ is a strong stateless simulation.
This is a routine proof by induction on m. □

As an alternative to the proof of $BreadthFirst(\langle\, s \,\rangle) \leqslant Search(\langle\, s \,\rangle)$ of Section 7.4.3, we could use transitivity of $\leqslant$ and combine Lemma 7.4.9 with a proof of $BreadthFirst(\langle\, s \,\rangle) \leqslant ParBF(\langle\, s \,\rangle)$.

Lemma 7.4.10 $BreadthFirst(\langle\, s \,\rangle) \leqslant ParBF(\langle\, s \,\rangle)$

Proof
Let $\mathcal{R} = \{(BFVisit(i, u, \overline{w}), ParBFVisit(i, u,) \parallel \Pi_{j=1}^{q} ParBF(\overline{w_j})) \mid i \geq 1, 1 \leq u \leq n,$
$\overline{w} \in \#_{j=1}^{q} w_j\}$. It is straightforward to show that $\mathcal{R}$ is a strong stateless simulation. □

We now arrive at the refinement ordering

$$\mathit{BreadthFirst}(\langle\, s \,\rangle) \leqslant \mathit{ParBF}(\langle\, s \,\rangle) \leqslant \mathit{Search}(s)$$

7.4.5 Some Further Refinements

Further refinements can be derived using the algebraic laws from Sections 4.4.2 and 4.4.3. Because a shorter path to v via u can be found only if there is an edge between u and v, we only need to look for shorter paths from u to neighbours of u; i.e. the vertices v such that $L(u, v) < \infty$. Formally, this depends on the following property:

$$x + L(u, v) < y \Rightarrow L(u, v) < \infty$$

By Corollary 4.4.38 from Section 4.4.3 then follows

$$(\Pi_{v=1}^{n} L(u, v) < \infty \triangleright \mathit{find}(u, v) \rightarrow \mathit{Search}(v)) \sim (\Pi_{v=1}^{n} \mathit{find}(u, v) \rightarrow \mathit{Search}(v))$$

hence $\mathit{Search}'(s) \sim \mathit{Search}(s)$ where

$$\mathit{Search}'(u) \mathrel{\widehat{=}} (\Pi_{v=1}^{n} L(u, v) < \infty \triangleright \mathit{find}(u, v) \rightarrow \mathit{Search}'(v))$$

If applied at this early stage of the derivation, this optimization persists throughout the subsequent stages of refinement. It has been omitted to keep the presentation of those later refinements simple. Alternatively, this optimization can be applied analogously at any later stage in the development of coordination strategies.

7.4.6 Concluding Remarks

Figure 7.4 illustrates the method of refinement. It shows how the schedules are related by the notions of refinement.

Figure 7.4: Lattice of Refinements of Shortest Path Schedules

7.5 Solving Triangular Systems

Finding the solution vector $\mathbf{x}$ of a triangular system

$$L\mathbf{x} = \mathbf{b} \tag{7.48}$$

where L is a triangular $N \times N$ matrix and $\mathbf{b}$ an N-dimensional vector, is a problem that arises in many application areas – for instance, when a system of equations is solved using Gaussian elimination or using an iterative method with an incomplete factorization type of preconditioner. Furthermore, in many applications, such as finite element applications, solving initial value problems by implicit methods, and solving non-linear equations using Newton-like methods, the system needs to be solved for multiple right-hand sides. In order to be able to solve complete systems efficiently on a parallel computer, both the factorization and the triangular solves need to be computed in parallel.

In this section, we apply our method of design to the problem of solving triangular systems. Without loss of generality only lower triangular matrices with a unit diagonal are considered.

In Section 7.5.1 we present a Gamma program for solving triangular systems and prove its correctness in Section 7.5.2. Subsequently, we derive a number of coordination strategies for this Gamma program in Section 7.5.3.

7.5.1 A Gamma Program for Solving Triangular Systems

We start by defining a multiset-based data representation. Inputs to this problem are the lower triangular matrix L and vector $\mathbf{b}$. In the initial multiset, L is represented by the following collection of tuples (called "$\mathcal{V}$"):

$$(\mathcal{V}, i, j, -l_{i,j}) \quad \text{for all} \quad 1 \leq j < i \leq N \tag{7.49}$$

Next, we model the solution vector $\mathbf{x}$. If $\mathbf{x}^*$ is a solution of (7.48) then its elements x_i $(1 \leq i \leq N)$ satisfy

$$x_i^* = b_i + \sum_{j=1}^{i-1} -l_{i,j} * x_j^* \tag{7.50}$$

We use the tuples $(\mathcal{X}, s, i, x_i)$ for $1 \leq i \leq N$ to represent the elements of $\mathbf{x}$. Here i denotes the index of the value x_i in $\mathbf{x}$. The field denoted s is used to count down the number of inner-product terms that still has to be added to b_i (from (7.50) follows that

this number initially equals $i-1$). The contribution of b_i to the solution is taken into account by initializing $x_i = b_i$, i.e. initially

$$(\mathcal{X}, i-1, i, b_i) \text{ for all } 1 \leq i \leq N \tag{7.51}$$

Formula (7.50) suggests that the problem can be solved using three basic computations: multiplying $-l_{i,j} * x_j^*$ (for all $j : 1 \leq j < i$); summing such product terms together, and adding them to b_i (for each $i : 1 \leq i \leq N$). Products $-l_{i,j} * x_j$ are computed by the following multiset rewrite rule $TS1$ which stores the result in an auxiliary tuple of the format $(\mathcal{Z}, r, i, z)$.

$$TS1 = (\mathcal{V}, i, j, v), (\mathcal{X}, 0, j, x) \mapsto (\mathcal{Z}, 1, i, v * x), (\mathcal{X}, 0, j, x)$$

Here i denotes the index of $\mathbf{x}$ to which the partial inner-product z belongs.

For summing the inner-products, we re-use the Gamma program for summation that we have already seen in Section 7.1. Rewrite rule $TS2$ extends the addition rule such that it keeps track of some additional administration: the index i indicates to which equation a (sum of) term(s) belongs, and the counter r which denotes the number of terms that has been aggregated.

$$TS2 = (\mathcal{Z}, r, i, z), (\mathcal{Z}, r', i, z') \mapsto (\mathcal{Z}, r + r', i, z + z')$$

The partial sums contained in the $\mathcal{Z}$-tuples are transferred to the solution vector by multiset rewrite rule $TS3$.

$$TS3 = (\mathcal{Z}, r, i, z), (\mathcal{X}, t, i, x) \mapsto (\mathcal{X}, t - r, i, x + z)$$

Note that the rewrite rules $TS2$ and $TS3$ compete for $\mathcal{Z}$-tuples. A partial result that has been gathered in some Z-tuple may be added to the solution vector, by $TS3$, before the complete inner-product is computed. Alternatively, inner-product terms may be aggregated, using $TS2$, before being transferred to the solution.

A Gamma program that solves triangular systems of linear equations consists of the parallel composition of these rewrite rules.

$$TS = TS1 + TS2 + TS3$$

Even though the design of this program was guided by a mathematical specification, its correctness is yet to be established formally. This is the subject of the next section.

7.5.2 Correctness of the Gamma Program

A specification of the required output of a program is called that program's postcondition. The correctness of a Gamma program can be established by showing that it satisfies a number of properties which together imply the postcondition. A good start for deducing properties of the output of a Gamma program is by calculating its termination condition. In addition to the termination condition, it may be necessary to find a suitable collection of invariants such that these properties together imply the postcondition.

We use the program logic of Chapter 2 to prove that the Gamma program TS correctly computes the solution to triangular systems of linear equation. To this end, we will propose a postcondition and show that this condition is met by the output of the program TS.

In the specification of the postcondition and in subsequent program properties, we use the following abbreviations for counting the different kinds of tuples.

$$\begin{array}{lcl}
\#\mathcal{X}(i) & = & (\#s, x : (\mathcal{X}, s, i, x)) \\
\#\mathcal{X} & = & (\#s, i, x : (\mathcal{X}, s, i, x)) \\
\#\mathcal{Z}(i) & = & (\#t, z : (\mathcal{Z}, t, i, z)) \\
\#\mathcal{Z} & = & (\#t, i, z : (\mathcal{Z}, t, i, z)) \\
\#\mathcal{V} & = & (\#i, j, v : (\mathcal{V}, i, j, v)) \\
\#\mathcal{V}(i) & = & (\#j, v : (\mathcal{V}, i, j, v)) \\
\#\mathcal{V}(i, j) & = & (\#v : (\mathcal{V}, i, j, v))
\end{array}$$

We will establish total correctness by proving the following

1. If TS terminates in a final states M, then M satisfies the postcondition $[\![Post]\!]M$, where $Post$ is defined by

$$(\forall i : 1 \leq i \leq N : (\#s, i, x : (\mathcal{X}, s, i, x)) = 1) \tag{7.52}$$

$$(\forall i : 1 \leq i \leq N \wedge (\mathcal{X}, s, i, x_i) : s = 0 \ \wedge \ x_i = x_i^*) \tag{7.53}$$

2. Given the initialization described, program TS terminates.

Generally, the postcondition of a Gamma program falls into two parts: an existential

part and a universal part. The existential part states that certain elements are present in the multiset and the universal part states that these elements represent a solution.

For the triangular solves program property (7.52) is the existential part of the postcondition. It requires that there are $\mathcal{X}$-tuples present in the multiset that represent a (solution) vector $\mathbf{x}$ of dimension N. Property (7.53) is the universal part of the postcondition. It states that x_i of $(\mathcal{X}, 0, i, x_i)$ equals the i^{th} component x_i^* of the actual solution vector $\mathbf{x}^*$.

The existential part (7.52) of the postcondition can be established by showing that it is invariant; i.e. there are always tuples in the multiset that represent the solution vector.

Lemma 7.5.1 *invariant* $(\forall i : 1 \leq i \leq N : \#\mathcal{X}(i) = 1)$

Proof

initially : Immediately from (7.51).

stable : $TS2$ leaves the $\mathcal{X}$-tuples unchanged. $TS1$ and $TS3$ reinsert an $\mathcal{X}$-tuple with the same i index as they consume. □

We continue by proving that property (7.53) of the postcondition holds at termination. To this end, we will use some invariants and the termination condition of the program. First, we will prove that, at termination, $s = 0$ for all $\mathcal{X}$-tuples. Subsequently, we will prove that if $s = 0$, then $x_i = x_i^*$.

To prove that, at termination, $s = 0$ for all $\mathcal{X}$-tuples we use Lemma 7.5.2 which shows how the value of s in $\mathcal{X}$-tuples is related, at any stage during execution, to the other elements in the multiset.

The value of s in $(\mathcal{X}, s, i, x_i)$ represents the number of inner-product terms $-l_{i,j} * x_i$ (see (7.50)) that still has to be added to the i^{th} element of the solution vector. The next invariant proves that this amount is equal to the sum of the number of partial inner-products that have already been computed, $(+t, i, z_i : (\mathcal{Z}, t, i, z_i) : t)$, plus the number of $l_{i,j}$'s for which the product $-l_{i,j} * x_i^*$ still has to be computed.

Lemma 7.5.2 *invariant* $\forall i : 1 \leq i \leq N :$
$(\forall s, x_i : (\mathcal{X}, s, i, x_i) : s = \#\mathcal{V}(i) + (+t, z_i : (\mathcal{Z}, t, i, z_i) : t))$

Proof

Initially : The result follows immediately from initialisation.

Stable : We show that the invariant is preserved by execution of any of the rewrite rules.

- $TS1(i,j)$: $\#\mathcal{V}(i)$ decreases by 1. All $\mathcal{X}$-tuples remain unchanged. A new $\mathcal{Z}$-tuple $(\mathcal{Z}, 1, i, v_{i,j} * x_i)$ is inserted, which increases the sum $(+t, z_i : (\mathcal{Z}, t, i, z_i) : t)$ by 1, thus retaining the equivalence.

- $TS2(i)$: Clearly all $\mathcal{X}$- and $\mathcal{V}$-tuples are unchanged. The replacement of $(\mathcal{Z}, r, i, z_i)$ and $(\mathcal{Z}, s, i, z_i')$ by $(\mathcal{Z}, r+s, i, z_i + z_i')$ leaves the sum $(+t, z_i : (\mathcal{Z}, t, i, z_i) : t)$ unchanged.

- $TS3(i)$: $\#\mathcal{V}(i)$ remains unchanged. Due to the removal of $(\mathcal{Z}, r, i, z_i)$, the term $(+t, z_i : (\mathcal{Z}, t, i, z_i) : t)$ is decreased by r. The value of the field, t, in $(\mathcal{X}, t, i, x_i)$ is also decreased by r, thus maintaining the equivalence.

□

The termination condition of $TS3$ is given by (7.54). It yields that there are no $\mathcal{Z}$-tuples at termination. Hence, at termination the summation of the t-values of the $\mathcal{Z}$-tuples (referred to in Lemma 7.5.2) equals zero.

$$\forall i : 1 \leq i \leq n : \#\mathcal{Z}(i) = 0 \vee \#\mathcal{X}(i) = 0 \tag{7.54}$$

Now, by Lemma 7.5.1 follows $(\forall i : 1 \leq i \leq n : \#\mathcal{Z}(i) = 0)$.

The termination condition of $TS1$ is given by (7.55). It gives information about the $\mathcal{V}$-tuples at termination.

$$\begin{array}{l}(\forall i, j : \#\mathcal{V}(i,j) = 0) \vee \\ (\forall i, j : \#\mathcal{V}(i,j) \neq 0 : (\forall s, x : (\mathcal{X}, s, j, x) : s \neq 0)\end{array} \tag{7.55}$$

We prove that, at termination of the program, the second term of the termination condition of $TS1$ cannot hold. As a consequence $(\forall i, j : \mathcal{V}(i,j) = 0)$ holds.

Suppose $\#\mathcal{V}(i,j) \neq 0$ for some i, j. Then let i' be the minimal i for which there exists a j' such that $\#\mathcal{V}(i', j') \neq 0$. By (7.49) follows $j' < i'$. Hence, since i' is the smallest i for which a $\mathcal{V}$-tuple exists, $\#\mathcal{V}(j') = 0$. From the termination condition of $TS3$ followed $\#\mathcal{Z} = 0$. Then by Lemmas 7.5.1 and 7.5.2 follows that there is a tuple $(\mathcal{X}, s, j', x)$ with $s = 0$. But this implies that $TS1(i', j')$ is enabled, which contradicts the assumption that the program has terminated.

Hence, at termination holds $\#\mathcal{Z} = 0$ and $\#\mathcal{V} = 0$. Then, by Lemma 7.5.2 follows $(\forall i : 1 \leq i \leq N \wedge (\mathcal{X}, s, i, x) : s = 0)$.

We proceed by showing that the value of x_i in the solution vector remains unchanged once $s = 0$. To this end, we first show some straightforward lowerbounds on the number of tuples of some type present in the multiset. Subsequently, we show that $s = 0$ implies $x_i = x_i^*$.

Lemma 7.5.3

1. *invariant* $\#\mathcal{V}(i) \geq 0$
2. *invariant* $\#\mathcal{Z}(i) \geq 0$
3. *invariant* $\#\mathcal{V} \geq 0$
4. *invariant* $\#\mathcal{Z} \geq 0$

Proof All cases follow from the observation that a number of tuples in the multiset can not be negative. □

The next lemma states that if at some stage there are no $\mathcal{V}$-tuples, then this will be the case for the remainder of the exection.

Lemma 7.5.4

1. $\forall i : 1 \leq i \leq N$ *stable* $\#\mathcal{V}(i) = 0$
2. *stable* $\#\mathcal{V} = 0$

Proof Both cases follows from the fact that there is no rewrite rule which introduces $\mathcal{V}$-tuples. □

For $\#\mathcal{V}(1)$ and $\#\mathcal{Z}(1)$ we can derive slightly stronger results which will be of use in future proofs.

Lemma 7.5.5 *invariant* $\#\mathcal{V}(1) = 0$

Proof
Initially : By initialization.
Stable : By Lemma 7.5.4.1.
□

Lemma 7.5.6 *invariant* $\#\mathcal{Z}(1) = 0$

Proof
Initially : By initialization
Stable : Execution of one of the rewrite rules $TS1(i,j)$, $TS2(i)$ or $TS3(i)$ with $i \neq 1$ leaves $\#\mathcal{Z}(1)$ unchanged. We consider the execution of rules $TS1(1,j), TS2(1)$ and $TS3(1)$.

- From Lemma 7.5.5 follows that $TS1(1,j)$ can not execute.
- From $\#\mathcal{Z}(1) = 0$ follow that both $TS2(1)$ and $TS3(1)$ are disabled.

□

Lemma 7.5.7 proves that the value x of tuples $(\mathcal{X}, s, i, x)$ does not change if $s = 0$. This ensures that once all updates have been performed, the solution will not change.

Lemma 7.5.7 $\forall i : 1 \leq i \leq N \wedge (\mathcal{X}, s, i, x_i) \wedge s = 0 :$ *stable* $x_i = k_i$

Proof Assume that $s_i = 0$ and $x_i = k_i$ for some tuple $(\mathcal{X}, s, i, x_i)$. We show that it still holds after execution of any of the rewrite rules from the program TS.
From Lemmas 7.5.2 and 7.5.3 follows $\#\mathcal{V}(i) = 0$ and $\#\mathcal{Z}(i) = 0$. Rule $TS1(i,j)$ leaves $\mathcal{X}$-tuples unchanged. Rules $TS2(i)$ and $TS3(i)$ are disabled because $\#\mathcal{Z}(i) = 0$. □

As an intermediate result, we show that the "lower triangular shape" of the $\mathcal{V}$-matrix implies that, for all $\mathcal{V}$-tuples, the column index j is smaller than the row index i.

Lemma 7.5.8 *invariant* $\forall i, j : (\mathcal{V}, i, j, v) : j < i$

Proof
Initially : By initialization.
Stable : None of the rewrite rules modifies a $\mathcal{V}$-tuple. □

We proceed by presenting an invariant which relates the x_i's of the $\mathcal{X}$-tuples to the x_i^* of the solution vector $\mathbf{x}^*$. In particular, this invariant tells us the value of the x_i's at termination.

In this invariant, the inner-products $-l_{i,j} * x_j$ associated with the $\mathcal{V}$-tuples that are still present in the multiset, hence still need to be added to the solution, are compensated for by the VX_i terms. Similarly, the partially aggregated inner-products that are (temporarily) stored in the $\mathcal{Z}$ tuples are accounted for by the ZS_i terms.

Lemma 7.5.9 *invariant* $\forall i : 1 \leq i \leq N : x_i^* = x_i + VX_i + ZS_i$

where $x_i = x \text{ in } (\mathcal{X}, s, i, x)$

$VX_i = (+\, j : 1 \leq j < i \wedge (\mathcal{V}, i, j, v_{i,j}) : v_{i,j} * x_j^*)$

$ZS_i = (+\, i' : 1 \leq i' \leq N \wedge (\mathcal{Z}, r, i', z_{i'}) : z_{i'})$

Proof

Initially : $\#Z = 0$, hence $ZS_i = 0$ for all i. Furthermore the x_i's are initialized $x_i = b_i$. Then the invariant reads $\forall i : 1 \leq i \leq N : x_i* = VX_i + b_i$. This holds by (7.50).

Stable : We consider each of the rewrite rules.

- We show that the invariant holds after execution of $TS1(i, j)$ by induction on i.
  - $i = 1$: By Lemma 7.5.1 follows that there always exists one tuple $(X, s_1, 1, x_1)$. This tuple is initialized at $x_1 = b_1 = x_1^*$ and $s_1 = 0$. By Lemma 7.5.7 follows *stable* $s_1 = 0 \wedge x_1 = x_1^*$. By Lemmas 7.5.5 follows $\#\mathcal{V}(1) = 0$, hence $VX_1 = 0$. By Lemma 7.5.6 follows $\#\mathcal{Z}(1) = 0$, hence $ZS_1 = 0$. Then the invariant holds for $i = 1$.
  - $i = k$: Assume the invariant holds for all $i : 1 \leq i < k$. Execution of $TS1(k, j)$ retrieves $(\mathcal{V}, k, j, v_{k,j})$ and $(\mathcal{X}, s_j, j, x_j)$ where $s_j = 0$. From $s_j = 0$ follows, by Lemma 7.5.2, that $\#\mathcal{V}(j) = 0$ and $\#\mathcal{Z}(j) = 0$. Furthermore, from Lemma 7.5.8 follows $j < k$, hence by the induction hypothesis we have $x_j = x_j^*$.
    The removal of $(\mathcal{V}, k, j, v_{k,j})$ decreases VX_k by $v_{k,j} * x_j^*$. Insertion of $(\mathcal{Z}, 1, k, v_{k,j} * x_j)$ increases ZS_k by $v_{k,j} * x_j$ where $x_j = x_j^*$. Hence, the increase of ZS_k cancels the decrease of VX_k and thus preserves the invariant.

  Hence, rewrites by $TS1(i, j)$ preserve the invariant.

- $TS2(i)$: All $\mathcal{X}$- and $\mathcal{V}$-tuples are unchanged, hence x_i and VX_i remain unchanged. The replacement of $(\mathcal{Z}, r, i, z_i)$ and $(\mathcal{Z}, s, i, z_i')$ by $(\mathcal{Z}, r + s, i, z_i + z_i')$ leaves ZS_i unchanged. Hence, rewrites by $TS2(i)$ preserve the invariant.

- $TS3(i)$: A substitution $\{(\mathcal{X}, t - r, i, x_i + z_i)\}/\{(\mathcal{Z}, r, i, z_i), (\mathcal{X}, t, i, x_i)\}$ decreases ZS_i by z_i and increases x_i from $(\mathcal{X}, s, i, x_i)$ with the same amount. Hence, rewrites by $TS3(i)$ preserve the invariant.

□

Earlier, we deduced from (7.54) and (7.55) that at termination $\#\mathcal{Z} = 0$ and $\#\mathcal{V} = 0$. This implies $ZS_i = 0$ and $VX_i = 0$ for all i. Hence, by Lemma 7.5.9 follows $x_i = x_i^*$ for all i. This concludes the proof of property (7.53).

Finally, we prove that the program TS terminates. To this end, we show that $T = (\#\mathcal{V}, \#\mathcal{Z})$ is a metric for the program TS.

Lemma 7.5.10 *$T = (\#\mathcal{V}, \#\mathcal{Z})$ is a metric under lexicographical ordering.*

Proof We write $\prec$ to denote the lexicographical ordering on $\mathbb{N} \times \mathbb{N}$. We show that execution of an arbitrary rewrite rule decreases T.

Let M and M' be multisets such that $M' = M[\sigma]$ where σ is a substitution of one of the rewrite rules from the program TS. Let $T = (v, z)$ in M and $T' = (v', z')$ in M'. We consider the possible cases for σ.

- σ is a substitution of $TS1$: Then $v' = v - 1$ and $z' = z + 1$, hence $(v', z') \prec (v, z)$,
- σ is a substitution of $TS2$ or $TS3$: Then $v' = v$ and $z' = z-1$, hence $(v', z') \prec (v, z)$.

□

By Lemma 7.5.3, both $\#\mathcal{V}$ and $\#\mathcal{Z}$ have a lowerbound of 0. Hence T has a lowerbound $\perp = (0, 0)$. Execution of any rewrite rule from TS decreases T, but T can not be decreased below $(0, 0)$. Therefore, TS must terminate.

7.5.3 Coordination Strategies

The correctness of the Gamma program TS implies that a method for solving triangular systems can be defined in terms of the rewrite rules $TS1$, $TS2$, and $TS3$. However, the program TS does not specify any particular order in which these actions should be performed. Hence, the program TS may behave as the most efficient, as a moderately efficient, as well as an inefficient triangular solve method.

In this section, we will derive several coordination strategies for the program TS. These strategies steer the execution of the program such that it exploits the available parallelism. The correctness of these coordination strategies follows from their derivation through a number of correctness preserving refinements. We show that some of these coordination strategies correspond to known algorithms for solving triangular systems.

We will use the general solution principle of divide and conquer to derive several coordination strategies for the triangular solves program. To this end we first decompose

the problem of solving triangular systems into subproblems. Solving some of these subproblems requires the solving of smaller triangular systems, to which we apply the same method recursively.

For the sake of generality, we consider a triangular matrix which is parameterized by a lower bound l and an upper bound h (hence it contains nonzero-elements for indices $i, j : l \leq j \leq i \leq h$). To cater for the application of the rewrite rules to subsets of the data in the multiset, we write $TSi_{l,h}$ to denote the strengthening of rewrite rule TSi where the reaction condition has been extended with the conjunct "$\wedge\, l \leq j < i \leq h$", and write $TSi_{l,m,h}$ for TSi extended with the conjunct "$\wedge\, l \leq j \leq m < i \leq h$."

Constructing the most general schedule for the program TS (parameterized by bounds l and h) yields

$$G_{l,h} \mathrel{\widehat{=}} !(TS1_{l,h} \rightarrow G_{l,h} \,\|\, TS2_{l,h} \rightarrow G_{l,h} \,\|\, TS3_{l,h} \rightarrow G_{l,h})$$

For any value $m : l < m < h$, the triangular matrix can be partitioned in three submatrices A, B and C as shown in Figure 7.5. The system of equations can be solved by first performing the computations associated with submatrix A, then those of B and finally those of C. The computations for submatrices A and C constitute the solving of a (smaller) triangular system, hence we can use the same solution strategy, and apply it to a smaller instance of the problem.

Figure 7.5: Decomposition of Triangular Solve

We will proceed as follows: first, we will prove that the most general schedule may be refined by a schedule which performs the computations in the ordering required by the ABC-decomposition.

Subsequently, we will pursue two alternative directions for further refinement. First we apply the same refinement recursively to the part of the schedule that takes care of

submatrix A. Secondly, we apply the strategy recursively to the part of the schedule responsible for submatrix C.

Formalizing the ABC-Decomposition

Since the computations associated with the submatrices A and C are again triangular solves, these can be described by $G_{l,m}$ and $G_{m+1,h}$. A schedule which describes the computations for block B is given by

$$T_{l,m,h} \quad \widehat{=} \quad !(TS1_{l,m,h} \to T_{l,m,h} \parallel TS2_{l,m,h} \to T_{l,m,h} \parallel TS3_{l,m,h} \to T_{l,m,h})$$

To formally justify that the most general schedule may be refined by a schedule that performs the computations in the order of the ABC-decomposition, we have to prove that (7.56) holds.

$$G_{l,m}; T_{l,m,h}; G_{m+1,h} \precsim_M G_{l,h} \tag{7.56}$$

We prove this using Lemma 4.3.21. This requires us to show the following:

i. $\mathcal{L}(G_{l,m}; T_{l,m,h}; G_{m+1,h}) \lhd \mathcal{L}(G_{l,h})$, and

ii. $\mathcal{O}(G_{l,m}; T_{l,m,h}; G_{m+1,h}, M) \subseteq \mathcal{O}(G_{l,h}, M)$.

Clearly, the components $G_{l,m}$, $T_{l,m,h}$, and $G_{m+1,h}$ consists of strengthenings of the rewrite rules of the program TS, hence obligation *i.* is met.

Next, we show property *ii.* by proving that every multiset in the output of $G_{l,m}; T_{l,m,h}; G_{m+1,h}$ satisfies the same postcondition as the multisets in the output of $G_{l,h}$. First, we establish the postcondition of $G_{l,h}$.

From the correctness proof of the program TS in Section 7.5.2 and Theorem 3.3.32 follows that for all $M' \in \mathcal{O}(G_{l,h}, M) : [\![post(G_{l,h})]\!]M'$ where

$$post(G_{l,h}) \quad \Leftrightarrow \quad (\forall s, i, x_i : l \leq i \leq h \wedge (\mathcal{X}, s, i, x_i) : s = 0 \ \wedge \ x_i = x_i^*) \tag{7.57}$$

$$(\forall i : l \leq i \leq h : \#\mathcal{Z}(i) = 0) \tag{7.58}$$

$$(\forall i : l \leq i \leq h : \#\mathcal{V}(i) = 0) \tag{7.59}$$

Next, we continue by showing that $G_{l,m}; T_{l,m,h}; G_{m+1,h}$ establishes the same postcondition. To this end, we first calculate the postcondition of $G_{l,m}$. Then calculate the postcondition of $T_{l,m,h}$ and combine this with the preceding to get the postcondition of

$G_{l,m}; T_{l,m,h}$. We combine this postcondition with that of $G_{m+1,h}$ to derive the postcondition of $G_{l,m}; T_{l,m,h}; G_{m+1,h}$.

We start with $G_{l,h}$. Firstly, we identify the precondition which ensures correct execution of a component $G_{l,h}$:

$$\begin{aligned} pre(G_{l,h}) \quad \Leftrightarrow \quad & \forall i : l \leq i \leq h : \#\mathcal{X}(i) = 1 && (7.60)\\ & \forall i : l \leq i \leq h : \#\mathcal{Z}(i) = 0 && (7.61)\\ & \forall i, j : j < l \leq i \leq h : \#\mathcal{V}(i,j) = 0 && (7.62)\\ & \forall i, j : l \leq j < i \leq h : \#\mathcal{V}(i,j) = 1 && (7.63)\\ & \forall s, i, x : l \leq i \leq h \wedge (\mathcal{X}, s, i, x) : s = i - l && (7.64)\\ & \forall s, i, x : l \leq i \leq h \wedge (\mathcal{X}, s, i, x) : x = x_i^* - \Sigma_{j=l}^{i-1} v_{i,j} * x_j^* && (7.65) \end{aligned}$$

By Theorem 3.3.32 follows that, because $G_{l,m}$ is a most general schedules for (an instance of) the triangular solve program, it establishes the same properties as the program TS (with bounds l and m). Hence, from the correctness proof in Section 7.5.2 and Theorem 3.3.32 follows that for all $M' \in \mathcal{O}(G_{l,m}, M) : [\![post(G_{l,m})]\!]M'$ where

$$\begin{aligned} post(G_{l,m}) \quad \Leftrightarrow \quad & (\forall s, i, x_i : l \leq i \leq m \wedge (\mathcal{X}, s, i, x_i) : s = 0 \ \wedge \ x_i = x_i^*) && (7.66)\\ & (\forall i : l \leq i \leq m : \#\mathcal{Z}(i) = 0) && (7.67)\\ & (\forall i : l \leq i \leq m : \#\mathcal{V}(i) = 0) && (7.68) \end{aligned}$$

Next, consider the postcondition established by $T_{l,m,h}$. By Theorem 3.3.32 follows that the negations of the reaction conditions of the constituent rewrite rules of $T_{l,m,h}$ hold. For $TS1_{l,m,h}$ this is given by (7.69).

$$(\forall s, j, x : l \leq j \leq m \wedge (\mathcal{X}, s, j, x) : s = 0 \Rightarrow (\forall i : m < i \leq h : \#\mathcal{V}(i,j) = 0)) \quad (7.69)$$

The termination condition for $TS2_{l,m,h}$ is subsumed by that of $TS3_{l,m,h}$ which is given by (7.70).

$$(\forall i : m < i \leq h : \#\mathcal{Z}(i) = 0) \quad (7.70)$$

In order to calculate the postcondition of $G_{m+1,h}$, we first verify that its precondition is established by execution of $G_{l,m}; T_{l,m,h}$. To this end, we observe that all properties of $post(G_{l,m})$ are stable, hence by Lemma 3.3.27 hold at termination of $G_{l,m}; T_{l,m,h}$.

Lemma 7.5.11 *If* $[\![pre(G_{l,h})]\!]M$, *then* $\forall M' \in \mathcal{O}(G_{l,m}; T_{l,m,h}) : [\![pre(G_{m+1,h})]\!]M'$.

Proof We verify the clauses of $pre(G_{m+1,h})$.

1. $\forall i : m+1 \leq i \leq h : \#\mathcal{X}(i) = 1$: Follows by invariant 7.5.1.

2. $\forall i : m+1 \leq i \leq h : \#\mathcal{Z}(i) = 0$: Follows by (7.70).

3. To prove $\forall i, j : j < m+1 \leq i \leq h : \#\mathcal{V}(i,j) = 0$ we consider the following cases:

   - $j < l$: From $pre(G_{l,h})$ follows (7.62). By Lemma 3.3.27 follows that (7.62) is not invalidated by the schedules $G_{l,m}$ and $T_{l,m,h}$.
   - $l \leq j \leq m$: At termination of $G_{l,m}$ holds (7.66). By Lemma 3.3.27 follows that (7.66) is not invalidated by $T_{l,m,h}$. Hence, the result follows from (7.69).

4. $\forall i, j : m+1 \leq j < i \leq h : \#\mathcal{V}(i,j) = 1$:
   From (7.63) follows that $\forall i, j : m+1 \leq j < i \leq h : \#\mathcal{V}(i,j) = 1$ holds initially. By Lemma 3.3.27 follows that this is not invalidated by the schedules $G_{l,m}$ and $T_{l,m,h}$.

5. $\forall s, i, x : m+1 \leq i \leq h \wedge (\mathcal{X}, s, i, x) : s = i - m - 1$: From case (3) and case (4) follows that $\forall i : m+1 \leq i \leq h : \#\mathcal{V}(i) = i - (m+1)$. Furthermore, by case (2), $\forall i : m+1 \leq i \leq h : \#\mathcal{Z}(i) = 0$. Hence the result follows by invariant 7.5.2.

6. $\forall s, i, x : m+1 \leq i \leq h \wedge (\mathcal{X}, s, i, x) : x = x_i^* - \Sigma_{j=m+1}^{i-1} v_{i,j} * x_j^*$:
   Follows from case (2) and invariant 7.5.9.

□

Next, we consider $G_{m+1,h}$'s postcondition. Analogous to $G_{l,m}$ follows that $G_{m+1,h}$ establishes $post(G_{m+1,h})$; i.e.

$$(\forall s, i, x : m+1 \leq i \leq h \wedge (\mathcal{X}, s, i, x) : s = 0 \ \wedge \ x_i = x_i^*) \tag{7.71}$$

$$(\forall i : m+1 \leq i \leq h : \#\mathcal{Z}(i) = 0) \tag{7.72}$$

$$(\forall i : m+1 \leq i \leq h : \#\mathcal{V}(i) = 0) \tag{7.73}$$

Now, we complete the proof of property *ii.* by showing that $G_{l,m}; T_{l,m,h}; G_{m+1,h}$ satisfies the same postcondition as $G_{l,h}$.

Lemma 7.5.12 $\forall M' \in \mathcal{O}(G_{l,m}; T_{l,m,h}; G_{m+1,h}, M) : [\![post(G_{l,h})]\!]M'$.

Proof We check that these predicates hold for all $M' \in \mathcal{O}(G_{l,m}; T_{l,m,h}; G_{m+1,h}, M)$.

- (7.58): At termination of $G_{l,m}$ holds (7.67). It is straightforward to show that this property is preserved by the subsequent schedules $T_{l,m,h}$ and $G_{m+1,h}$. Furthermore, at termination of $G_{m+1,h}$ holds (7.72). The result follows from (7.67) $\wedge$ (7.72).

- (7.59): Analogous to the previous case.

- (7.57): Follows from the preceding cases and invariant 7.5.9.

□

Now that we have shown that properties *i.* and *ii.* holds, we conclude that (7.56) indeed holds.

We continue the derivation of coordination strategies with two alternative subsequent refinement which exploit that the computations associated with submatrices A and C again constitute the solving of triangular systems. Hence, refinement (7.56) can be applied to the schedules for these submatrices.

Applying refinement (7.56) to both A and C yields a strategy that is of interest from an algorithmic point of view, however, does not lend itself well for parallel execution. Therefore we will not pursue that direction further. Instead we will examine the coordination strategies that result from repeated application of refinement (7.56) to either one of the schedules for submatrices A or C.

Derivation of an Inner-Product Coordination Strategy

In this Section we derive a coordination strategy for the triangular solves program TS that results from successively applying refinement (7.56) to the topmost triangular submatrix (A).

For successive refinements, we take a fixed value b to indicate the size of the block B (indicated by m in Figure 7.5). The component responsible for the area A can be refined as long as the sides of the area are at least b. Thus, we derive the coordination strategy $F_{l,b,h}$ defined by

$$F_{l,b,h} \quad \widehat{=} \quad 0 < b < h - l \triangleright F_{l,b,h-b}; T_{l,h-b,h}; G_{h-b+1,h}\,[G_{l,h}]$$

This derivation can be depicted as follows (note that this does not correspond to the order of computation which proceeds from top to bottom). The formal justification of the correctness of $F_{l,b,h}$ is given by Lemma 7.5.13.

Lemma 7.5.13 *For all* $M : [\![pre(G_{l,h})]\!]M : F_{l,b,h} \precapprox_M G_{l,h}$

Figure 7.6: Block Inner Product Strategy

Proof By induction on $h - l$.

- $h - l \leq b$: Then $F_{l,b,h} \approx_M G_{l,h}$.

- $h - l > b$: Then we reason as follows

$$
\begin{array}{ll}
 & F_{l,b,h} \\
\approx_M & \qquad \{ \text{ def. } F \ \} \\
 & F_{l,b,h-b}; T_{l,h-b,h}; G_{h-b+1,h} \\
\precapprox_M & \qquad \{ \text{ Ind. Hyp. \& Corollary 4.3.13 } \} \\
 & G_{l,h-b}; T_{l,h-b,h}; G_{h-b+1,h} \\
\precapprox_M & \qquad \{ \text{ (7.56) } \} \\
 & G_{l,h}
\end{array}
$$

□

The strategies resulting from these refinements can be categorized using BLAS terminology. BLAS is a library of basic linear algebra subprograms, for use in numerical computations. These subprograms are classified according to the type of operations: level 3 BLAS contains matrix-matrix operations, level 2 BLAS contains matrix-vector operations and level 1 BLAS contains vector-vector operations. A more elaborate description of this library can be found in, for instance, [49].

For $b > 1$, the component $T_{l,b,m}$ of the schedule $F_{l,b,h}$ describes a matrix-vector computation. According to the BLAS classification, this is a level 2 operation. For $b = 1$, the strategy $F_{l,1,h}$ describes a strategy in terms of vector-vector (inner-product) operations that is also known as the row sweep [56] or *ij*-method [104]. These are BLAS1 primitives.

The inner-product can be further refined, by performing the computations in a recursive doubling manner. In its turn, the recursive doubling strategy can be refined by a strategy which performs the computation in a sequential, element-wise fashion which can be considered BLAS0 operations.

Derivation of a Vector-Update Coordination Strategy

Next, we consider the alternative avenue of refinement where the schedule for the rightmost triangular area (C) is successively refined using (7.56). This derivation yields $H_{l,b,h}$ defined by

$$H_{l,b,h} \quad \widehat{=} \quad 0 < b < h - l \triangleright G_{l,l+b-1}; T_{l,l+b-1,h}; H_{l+b,b,h}\,[G_{l,h}]$$

In this case, the order of derivation, depicted below, corresponds to the order of computation.

Figure 7.7: Block Vector Update Strategy

To formally justify the correctness of $H_{l,b,h}$ Lemma 7.5.14 proves that it is a refinement of $G_{l,h}$.

Lemma 7.5.14 *For all* $M : [\![pre(G_{l,h})]\!]M : H_{l,b,h} \precapprox_M G_{l,h}$

Proof By induction on $h - l$.

- $h - l \leq b$: Then $H_{l,b,h} \approx_M G_{l,h}$.

- $h - l > b$: Then we reason as follows

$$\begin{array}{ll} & H_{l,b,h} \\ \approx_M & \qquad \{ \text{ def. } H \ \} \\ & G_{l,l+b}; T_{l,l+b,h}; H_{l+b+1,b,h} \\ \precapprox_M & \qquad \{ \text{ Ind. Hyp., Lemmas 7.5.11 \& 4.3.12 } \} \\ & G_{l,l+b}; T_{l,l+b,h}; G_{l+b+1,h} \\ \precapprox_M & \qquad \{ \ (7.56) \ \} \\ & G_{l,h} \end{array}$$

□

Taking $b > 1$ yields a strategy where the $T_{l,b,h}$ describes a matrix-vector multiplication; i.e. a BLAS level 2 operation. For $b = 1$, the matrix-vector multiplication reduces to vector-vector operations which are BLAS1 level operations. This strategy is known in the literature as the column sweep [56] or *ji*-method [104]. The vector update can be further refined by computing the vector products in a sequential fashion. Hence, this computation proceeds by element-wise (point-point) computations. In BLAS-terminology, these can be called level 0 BLAS operations.

7.5.4 Concluding Remarks

In this case study the Gamma and coordination models are used as intermediate levels in the process of program design. The Gamma program specifies the functionality by means of basic computations while abstracting from operational aspects. The coordination language is used to derive a high-level strategy (in terms of blocks of individual computations) for executing the Gamma program.

Figure 7.8 gives an overview of the derivation process of the coordination strategies for the triangular solves program and shows how the resulting strategies are related by the notion of refinement.

The main refinement in this section is justified by proving that it is a statebased refinement. It is interesting to note that this refinement was not proven by constructing a simulation relation. Instead, we used a technique which requires that the refining schedule yields the same output as the schedule that is being refined.

The output of the most general schedule for the triangular solves program is the same of the output of the corresponding program which was already proven in the correctness proof of the program.

The schedule that refines the most general schedule for solving triangular systems consists of the sequential composition of schedules that again have the form of most general schedules. A combination of techniques allowed us to compute the output of the refining schedule in a conventient manner: Firstly, there is a straightforward way of combining the outputs of sequentially composed schedules. Secondly, the fact that the subschedules were again most general schedules (but for smaller problems) allowed us to establish their postcondition in an practical manner: if a schedule again solves a triangular system as subproblem, then it satisfies (for a subdomain) the same postcondi-

Figure 7.8: Lattice of Refinements of Triangular Solve Schedules

tion as the original problem. Additionally, the postcondition of a most general schedule can, through the correspondence with its Gamma program, be established by taking the conjunction of the termination condition of the constituent rewrite rules.

This case study showed that, although statebased refinement is not a precongruent notion, it can in some cases be used in a practical fashion.

For comparision with a simulation-based proof we refer to [28]. There we prove the correctness of the vector-update and column-update strategies by providing statebased simulations. The construction of the simulation relations used there is fairly straightforward, and establishing the invariance of useful properties requires only simple inferences. However they also require the keeping track of a lot of details relating to the indices of the matrix and vector elements.

An alternative formal derivation of a triangular solve is described by Loyens and Bisseling [88]. They use predicate calculus and invariants to formally derive an algorithm for solving triangular systems. The resulting algorithm is described as the parallel composition of a number of (CSP-like [76]) parametrized processes.

They start with choosing a network topology and a data-distribution and base their derivation on this choice. The resulting algorithm is inseparably tied to this choice and cannot be adapted to other architectures.

This contrasts with our approach which yields algorithms that are independent from,

but can be mapped onto a variety of architectures. However at this high level of description, it is already possible to assess the potential performance of the algorithms. If we assume that the execution of a rewrite rule takes unit time, then the vector update algorithm can be executed in $\mathcal{O}(N)$ time and the inner-product algorithm can be executed in $\mathcal{O}(N \log N)$ time (where N is the dimension of the matrix).

In a subsequent implementation phase a choice may be made between different kinds of data-distributions. This provides the opportunity to select a data-distribution that matches the chosen algorithm with the available architecture.

A further difference between our approach and theirs is that they derive only a single algorithm whereas we derive a family of algorithms that is related by a notion of refinement.

Acknowledgement

The case study presented in this section is based on joint research with Arno van Duin [31], [32].

7.6 Evaluation of the Methodology

In this we evaluate the method of design (with emphasis on the development of the coordination component) and the usability of the notions of refinement from Chapters 4 and 6 in this process. Furthermore, we give general guidelines for applying the refinement techniques.

7.6.1 Overview of the Design Methodology

Based on the case studies in this chapter, we outline in this section general guidelines that may be followed for the design of programs.

Firstly, a Gamma program needs to be designed. This can be done using the "Discipline of Gamma programming" [12] which ensures the program's correctness by construction. Alternatively, the correctness of the program can be verified a posteriori using the programming logic of Chapter 2 as is illustrated in Section 7.5.

The next step is the derivation of a coordination strategy for this Gamma program. To this end the most general schedule should be constructed (using Definition 3.3.1[2]). This most general schedule forms the initial specification of the operational aspects of the program.

Subsequently, the derivation proceeds along the following lines.

1. Invariants of the program are used, through convex refinements, to make the ranges over which the variables of rewrite rules of the schedule vary explicit.

2. Decompositions: convex and statebased refinement may be used to introduce rewrite rules for all elements in the range of a variable of a rewrite rule. This effectively transforms nondeterminism in the selection of data into nondeterminism in the selection of rules.

3. Rearrange temporal ordering of rewrite rules using stateless simulation and stateless refinement laws. Specializing the temporal order of execution reduces the nondeterminism in the selection of rules.

The approach described above describes the design trajectories follows by Prime Sieving, Sorting, Shortest Paths and Solving Triangular Systems. In the derivation

[2]If the given Gamma program is not in a product of sums form, then Sands' results from [106] can be used to transform it into such a form.

of schedules for summation, the first phase is absent. However, for summation, the nondeterminism in the selection of data is not relevant to the algorithms. Consequently, for this case, there is no need to eliminate the nondeterminism in the selection of data.

7.6.2 Validation of Proof Methods for Refinement of Coordination

In Chapters 4 and 6 we introduced three kinds of notions of refinement: statebased, convex and stateless refinement. Each of these notions comes equipped with one or more methods for using it. In this section we describe the rôle each of these notions plays in the derivation of coordination strategies.

- Statebased refinement contains every refinement that follows from the operational semantics. However, because it is not a precongruence, the only proof methods available are strong and weak simulation (up-to).

  Therefore, statebased refinement is the safety-net of the collection of refinement notions: if no other notion justifies a refinement, then we resort to statebased refinement. The application of statebased simulation in the cases summation (Section 7.1), Ripple Sort (Section 7.3.3), Shortest Paths (Section 7.4) and Solving Triangular Systems (Section 7.5) shows that these are nonetheless a feasible proof method.

  An important reduction of the complexity of statebased simulation proofs is due to Lemma 3.2.5 (which is used in all of the cases mentioned). It allows reasoning about parallel schedules in terms of all possible interleaved behaviours, thus avoiding the combinatorial complexity of multi-step transitions.

  The main effort in devicing statebased simulations proofs consists of finding a general form that describes all forms that a schedule may evolve into during execution and finding a relation between the form of the schedule and (invariant) properties of the multiset. Such statebased simulation proofs resemble *wp*-style proofs (see e.g. [50] and [80]) in a way that is shown by the following example.

  Example 7.6.1 *Consider the following imperative program*

  $$\begin{array}{l} \texttt{for } k = 1 \texttt{ to } N \texttt{ do} \\ \quad \texttt{f}(k) \end{array}$$

where `f` *is some action. Suppose that this program satisfies an invariant* $I(k)$*. The same invariant can be used in a statebased simulation. First, we write the program as the schedule* $S(1)$ *where* $S(i)$ *is defined by*

$$S(i) \mathrel{\widehat{=}} i \leq N \triangleright f(i)$$

A statebased simulation relation $\mathcal{R}$ *which involves* S *could incorporate the invariant as follows*

$$\mathcal{R} = \{(\langle S(k), M\rangle, \langle \ldots, M\rangle) \mid [\![I(k)]\!]M\}$$

This example suggests that refinements based on the *wp*-calculus can be transformed into a statebased simulation proof.

- Convex refinement comprises a combination of the *wp*-style proofs (which use invariants that are related to a locus of control) with properties that are global to programs such as advocated by UNITY. It provides a new technique for reasoning about refinement in a setting which allows interference.

  In the generic framework of refinement that was presented in Chapter 5, convex refinement can be understood as striking a balance between statebased and stateless refinement because it can exploit properties of the multiset, while also giving rise to algebraic laws that may be applied in a modular equational style.

  The derivations of the coordination strategies for prime sieving (Section 7.2) and Selection Sort (Section 7.3.4) are almost completely justified using only the convex laws. This shows that the convex refinement laws are quite powerful.

  The logical properties that underlie (uniform) parallel and sequential loop structures have been formulated in a general way which caters for the introduction of such structures through the use of a single refinement law. Most likely there are many other kinds of logical properties which underlie different coordination structures. For example, the divide-and-conquer structure of Quicksort suggests that it may be justified by a uniform progression/collection of logical properties It may be possible to capture the introduction of such a coordination strategy using a single convex refinement law.

  Further investigation of the logical properties which underlie the introduction of coordination structures would be very interesting because it could give insight in generic solution strategies.

- Stateless refinement provides a wide range of algebraic laws. These laws may be used in a modular and equational style of reasoning about refinement (similar to the way that equals are substituted for equals in traditional mathematics). Furthermore, amongst the methods presented in this thesis, the stateless laws provide the highest level of abstraction from operational details. Because of these features, equational reasoning using the stateless laws is the most convenient method for reasoning about refinement. The fact that stateless refinement laws are used in all case studies confirms the high practical value of this method.

  Furthermore, Theorem 5.2.13 and Theorem 5.5.16 prove that the strong and weak stateless laws may be used in combination with other strong notion of refinement (including in up-to combinations). Consequently, stateless refinement can be used to simplify proofs that use simulation-based proof techniques.

  In the shortest paths case (Section 7.4), stateless refinement was used in an interesting way that differs from its application in the other cases. First, a stateless simulation was used, in Lemma 7.4.7, to proves a refinement of a coordination structure that is specific to the shortest paths application. In the remainder of the shortest paths case, this refinement could be used as a law in an algebraic manner.

  The absence of multisets in stateless simulations precludes the use of invariants for relating schedules and multisets of configurations (as is typical for statebased simulations). Consequently, stateless simulations only employ invariants based on the form of schedules.

Besides providing a more convenient method of reasoning, (in)equational refinement laws have another advantage over simulations. Refinement laws suggest possible ways in which a schedule may be refined; hence they can be used in a constructive manner to guide the derivation (this was, for example, the case for the refinement of Section 7.4.2 of the shortest paths case). In contrast, simulations can only be used for a posteriori verification of refinements.

Both the equational and simulation based methods can be supported by automated tools. Equational reasoning could be supported by theorem provers and simulation proofs could be facilitated using model-checkers. This would further alleviate the burden on the program engineer and reduce the opportunities for making errors.

8 Related Work

An essential feature of the method of program design that is presented in this thesis is that the correctness and complexity aspects are addressed by separate models for computation and coordination. Associated with these models are formalisms for reasoning about them.

In the next section we will discuss some other programming methods and mention some commonalities and differences with the method presented in this thesis. In the subsequent section we will discuss some formalisms for reasoning about parallel systems with shared memory and compare those to the methods we have used in this thesis.

The ubiquity of coordination throughout computer science has lead to a broad field of study. Some general starting points for this area of research are [1] and [35].

8.1 Separation of Computation and Coordination

According to the method proposed in this thesis a program can be seen to consist of a computation component and a (separate) coordination component. The computation component defines *what* a program computes and is responsible for correctness. The coordination component specifies *how* a program computes and thereby defines that program's time and space complexity.

In some guise or another correctness and complexity have been recognized as the most important aspects of programs throughout the academic computer programming community. Moreover, it has long been suggested that these aspects be dealt with separately. This has lead to the approach of "program derivation by successive step-wise refinement" for the most significant programming language paradigms.

We consider the major programming paradigms and examine to which degree computation and coordination aspects are separated in associated methods of program design.

8.1.1 Functional Programming

We give a brief introduction to the ideas that underlie the functional programming paradigm. A introductory text in this area is [16].

The idea behind functional programming is that a program can be seen as a function which transforms an input into an output. In accordance with this view, the making of a functional program consists of constructing an expression that denotes a function which relates outputs to inputs in the required manner. To facilitate the definition of such expressions, functional programming languages provide a collection of primitive functions (including constants) and a method for creating new functions by giving defining equations in terms of existing functions.

A computation consists of rewriting some initial expression by successively substituting one side of a defining equation of the program by the other side, until no more rewrites are possible. The resulting term is said to be a "normal form" of the initial expression.

This rewriting process has its theoretical foundations in the λ-calculus [14]. A theoretical result from the λ-calculus states that every (well formed) expression has a unique normal form. A consequence of this result is that the order in which an expression is rewritten is irrelevant to the result. In particular, disjoint subexpressions may be rewritten in parallel.

Hence in designing a functional program, the programmer does not need to concern himself with the operational aspects of his program and may focus on the correctness of the output it yields. As a secondary concern, the programmer may want to optimize the execution of his program. To this end, the functional programming community suggested the "transformation approach":

> The transformation approach is to separate the [...] programming task in two stages. Firstly to write the program concentrating on making it as clear and understandable as possible, neglecting efficiency considerations, and then to successively transform this to more and more efficient versions using methods guaranteed to preserve the intent of the original program while improving its efficiency [41].

The methods for transforming functional programs are based on the principle of replacing equals by equals. This principle is valid for functional programs because, in contrast to imperative programs, they satisfy the principle of "referential transparency" which states that variables always denote the same quantity (within the context in

which they are defined). As a result, it is possible to use the algebraic properties of the primitives of functional languages to transform programs in an equational style. A survey of transformation techniques for functional (and logic) programs is given in [95].

The approach described in [68] suggests that this arsenal of transformation techniques can be used to increase the efficiency of Gamma programs once these are expressed in a functional programming language.

However, it is rather unsatisfactory that in order to change the behaviour of a functional program one has to change its representation (i.e. the defining equations) which is promoted for its abstraction from operational aspects. Instead of their representation, it is some execution mechanism that is external to functional programs that actually defines their behaviour. The reason for transforming programs is to increase the degree in which the representation concords with the execution style of this mechanism. In this sense, the method of functional programming has in common with that of imperative programming that programs are adjusted to match an underlying machine (albeit a more abstract one than the Von Neumann machine towards which imperative programs are tailored).

This inadequacy of functional languages to express the behaviour of programs is recognized by Darlington in [42]. There, a language akin to temporal logic is used to give a separate specification of the behavioural aspects of a program which must be satisfied by the implementation.

As alternative approach for obtaining finer control over the behaviour of functional programs, the use of annotations has been suggested by many authors. For example, Hudak's "para-functional" programming [78], Nöcker et. al.'s Concurrent Clean [92] and Cox et. al.'s language Caliban [39] propose to extend functional programming languages with annotations which include primitives for process creation and termination, combinators for sequential and parallel composition and mappings of tasks to processors. In these cases the annotations are dispersed through the program text. In contrast, we propose in this thesis to specify the behaviour textually separate from the specification of the computation. This simplifies reasoning about operational aspects without reference to the computation aspects and facilitates algebraic manipulation of the behaviour without affecting the correctness of the computation.

8.1.2 Logic Programming

Logic programming is a paradigm of computation that is based on the idea that a computation can be seen as a series of inferences in a formal logic. Logic programs consist of inference rules in a first order language (for instance predicate logic) such that their solution can be obtained by an automated-proof procedure (in particular by resolution and pattern matching). General introductions to the area of logic programming are [87] and [2].

One of the earliest accounts of an approach towards programming where the computation and complexity aspects are addressed separately is [69]. In that paper Hayes argues that a logic program can be considered to consist of a collection of deduction rules (which can be used to generate proofs) and a theorem proving mechanism which controls the application of the deduction rules.

Building on Hayes' insights, Kowalski publishes [82] in which he writes.

> [...] when *logic* is separated from *control*, it is possible to distinguish what the algorithms does, as determined by the logic component, from the manner in which it is done, as determined by the control component.

Consequently, he advocates the following approach to the design of logic programs in [84].

> The problem of developing a correct but efficient program can usually be decomposed into two simpler subproblems:
>
> 1. Specification. The first task is to specify the problem to be solved and the information which is needed for its solution.
> 2. Efficiency. Inefficiencies implicit in the problem specification can then be identified and removed, transforming the specification into an effective program.

The phases of this method are supported by the following techniques

- A method for massaging a specification in predicate logic into the format required by logic programming languages (see Chapter 10 of [83]).
- A collection of correctness-preserving program transformations for increasing the efficiency of the program (based on the seminal work [20]; a recent overview is given in [95]).

These articles by Hayes and Kowalski suggest that the ideas for separately developing a program's computation and coordination aspects were present at an early stage of the evolution of logic programming. However, instead of putting this idea to use, research continued towards the design of autonomous proof procedures which would be sufficiently general to ensure satisfactory performance for all possible logic programs. This focus of attention was justified as follows

> The control component can be expressed by the programmer in a separate control language; or it can be determined by the program executor itself. The provision of a separate control language allows the programmer to advise the problem-solver about program execution and is suitable for the more experienced programmer. The determination of control by the program executor, on the other hand, relieves the programmer of the need to specify control altogether and is more useful for the inexperienced programmer, the casual database user, and even the expert programmer during the early stages of program development. (from p. 127 of [83])

Because logic programming languages were not designed for describing operational aspects, they were extended for parallel programming by adding explicit (extra-logical) mechanisms for synchronization and communication. Consequently, the computation and coordination aspects of (parallel) logic programs are intertwined in a single program text. Overviews of approaches towards parallel logic programming are given in [109] and [37]. As a result, there is no general method for reasoning solely about the behavioural aspects of logic programs. This becomes particularly desirable for programs which deviate from the default behaviour through the use of extra-logical features such as the cut operator or synchronization and communication primitives.

A relation between the Gamma model and logic programming is described in [33]. There, the Gammalög programming language is presented which shares features of both logic programming and the Gamma model. A sequential prototype of this language has been implemented using the programming language Gödel [71].

8.1.3 Imperative Programming

The functional and logic programming paradigms are based on the idea that a program is an abstract object that can be captured by specifying its mathematical properties from a particular perspective. In contrast, the imperative programming paradigm has

evolved as a method for instructing a machine to behave in a certain manner. Data is stored in variables which essentially are named addresses in the memory of a computer. The basic unit of computation is the assignment statement which stores a value in a variable. Programs are built by defining an order of executing assignments. To this end, the following control-flow constructs are typically employed: sequential compositions, conditional composition (if-then-else) and the iterative constructs (for, while and repeat).

The following quotation (from p. 237 of [61]) illustrates that in the imperative programming community the benefits of structuring the design of a program in phases is also recognized.

> The programmer has two main concerns: correctness and efficiency [...] When faced with any large task, it is usually best to put aside some of its aspects and concentrate on the others [...]. This important principle is called *separation of concerns*

The methodology for the design of sequential programs differs somewhat from those of the functional and logical programming paradigms because due to their lack of useful mathematical properties, imperative programs do not lend themselves well for transformation. However, a transformational method of program development can be realized if a formalism is used which is able to express both specifications and programs. We will discuss two of such formalisms which have also been put forward as methods for the design of parallel programs.

Action Systems

The action system formalism for parallel computing was introduced in [7, 8]. An action system consists of a set of guarded assignment statements that co-operate through shared variables. The statements of an action system may be executed in an arbitrary order; statements may be executed in parallel as long as the actions do not have any variables in common. Actions are executed atomically; i.e. without interference from any other action in the system.

Initially, a calculus of refinement of action systems was based on weakest preconditions [4, 5]. Based on this notion of refinement a step-wise method for the development of parallel programs is presented in [6, 9, 108]. Later, methods of refinement with a larger emphasis on the refinement of behavioural aspects of action systems were described in [113] and [10]. There, refinement is defined in terms of traces of events of action systems.

The action systems model has in common with Gamma that it consists of a collection of actions that is executed in an arbitrary order. However, the notions of refinement for action systems take sequential behaviour as the default and gradually introduce opportunities for parallel execution.

> [..] the goal is to transform a more or less sequential algorithm or algorithm specification into an action system that can be executed in a highly parallel fashion, so we need special refinement rules to introduce parallelism into the execution. (p. 122 [9]).

This is dual to the method of refinement proposed in this thesis which starts with parallel behaviour and gradually increases sequentiality.

Furthermore, the kind of refinement employed by action systems also differs from the one used in this thesis. Refinement of action systems is based on the idea of "action refinement"; i.e. an action gets replaced by a collection of actions that achieve (a refinement of) the same relation between input and output. When using this method of refinement, a program is developed from a specification by successively inventing the actions that can be used to implement a specification. The method of action refinement implies that both computation and coordination issues are resolved in a single refinement step.

Notwithstanding these differences, there are some efforts in the action systems community which tend toward the separate development of computation and coordination aspects.

For instance, in [8] a formal method for decentralizing the control of a given action system is described. Decentralization effectively consists of decomposing a single action system into a layered system where a higher layer controls the order of computations of lower layers (without interfering with the computations). On a conceptual level, this is related to the coordination approach.

A benefit of this approach is that it straightforwardly caters for multiple layers of coordination which can be thought of as higher-order coordination. A draw-back of this approach is that the coordination is encoded by means of control-variables which are used to synchronize the actions and transport values back and forth between components of an action system. This method is not designed for the specification of coordination strategies and, according to [108], results in tedious verification steps.

The details of refinements can be suppressed by using a collection of transformation laws. Such a collection of laws for the transformation of sequential action systems into

parallel action systems is described by Sere in [108].

In [70], Hedman, Kok and Sere propose a structured method for the refinement of action systems. This approach is based on specifying action systems as the (prioritized) composition of a computation part and a coordination part. Following this discipline, the computation and coordination parts can, to a large extent, be refined independently. However, because coordination is achieved through shared variables, a data-refinement in either the computation part or the coordination part requires that additional conditions be checked in both parts with respect to these variables.

Unity

The UNITY framework, introduced by Chandy and Misra in [23], consists of a programming language and a programming logic. A UNITY program consists of a set of guarded (multiple) assignment statements. The execution mechanism resembles that of action systems and Gamma in that this set of statements is repeatedly executed in a nondeterministic order until a fixed-point is reached.

The programming logic is based on a small set of temporal properties (of sequences of states). UNITY's approach to program development is to start with a specification (in predicate logic) and successively introduce more concrete actions by action refinement. A typical example of such a refinement is that a (assignment) statement is replaced by a collection of statements that operate upon a more detailed representation of the data but satisfy the same temporal properties. The relation of this approach to the approach followed in this thesis is essentially the same as that described for action systems.

A methodological difference between the UNITY and action systems approaches is that UNITY is aimed at refining specifications, while action systems are aimed at refining programs. This distinction has become less strict by the extension of the UNITY method with the structuring mechanism of procedures and local variables in [111].

The nondeterministic execution mechanism of UNITY makes it in principle susceptible to the superposition of a coordination component. However, such an approach is not taken (nor could it be found in the current literature) and the ordering of execution of UNITY programs is defined by operators for sequential composition, (may) parallel composition and synchronous (must) parallel composition.

This obstructs the mutually separate refinement of the computations and coordination and prevents the re-use of coordination components for computation components with similar structure.

Concluding Remarks

The approaches of both action systems and UNITY introduce an ordering on the computation by incorporating coordination structures in the same program text. This means that programs have to be redesigned for different architectures.

Furthermore, the use of the imperative style of representing data by variables (rather than tuples) imposes premature constraints on the execution. As a consequence, special attention has to be paid to the discovery of potential parallelism during the process of refinement.

8.2 Reasoning about Parallel Shared Memory Programs

A large portion of this thesis is devoted to the development of formal methods for reasoning about the correctness and refinement of parallel programs which share a common memory. In this section we survey some other formal methods that have been proposed for reasoning about the correctness of such programs. In particular we examine in what way they support the development of programs by step-wise refinement.

We classify the methods according to the style of semantics that is used. We do not claim that these categories are disjunct - some methods may fit in more than one category.

8.2.1 Axiomatic/Assertional Reasoning

In [73] Hoare suggested the use of assertions to reason about (partial) correctness of sequential programs. This method uses triples $P\ \{S\}\ Q$, where P and Q are predicates and S is a statement, to mean that, if S is executed when pre-condition P holds, then post-condition Q holds if S terminates.

Starting from an axiom that defines the effect of an assignment, the effect of a program can be derived from its components using a set of inference rules for all combinators in the programming language.

Extensions of this approach for dealing with primitives for coordinating concurrent execution or mutual exclusion of parallel programs have been proposed; e.g. in [75], [93], [85]. These extensions have in common that they are based on a formal notion of non-interference. Effectively, these methods require that the proof of a program be

formulated in terms of properties that are interference-free. These are essentially the same kind of properties that are used for proving convex refinements.

The action systems approach is also essentially an assertional method. This approach is closely related to the methods of imperative program design that we discussed in Section 8.1.3.

An axiomatic method for reasoning about Gamma programs is presented in [52] and [51]. This is based on the axiomatization of the pre- and post condition of the rewrite rules of Gamma programs. In composing these conditions, this approach takes interference into account by allowing the precondition of a transition to differ from the postcondition of the preceding transition. This corresponds to the way in which interference is modelled in the framework of generic refinement. This approach yields a compositional method for reasoning about input-output refinement of Gamma programs. However, the method tends to be impractical for manual proofs and provides little support for the design of programs by step-wise refinement.

A systematic approach to the construction of an axiomatic-style program logic for Gamma is pursued in [57]. The programme of that paper is to derive a program logic by applying Abramsky's domain theory in logical form to a (denotational) resumption semantics for Gamma.

8.2.2 Denotational Methods

The aim of the denotational semantics is to find ways of defining the meaning of programs in a compositional manner; i.e. methods for calculating the meaning of a program from the meanings of its constituent parts.

In the denotational style, several methods based on the concept of transition traces have been proposed for assigning meanings to programs that communicate asynchronously via shared memory. We will discuss some of these methods in this section.

For an imperative parallel language with shared memory, Brookes presents in [19] a number of laws, based on transition traces, which resemble some of the stateless laws that we presented in Section 4.4.2. These laws have in common that they are independent from the current state. This correspondence suggests that Brookes' law $C_1; (C_2 \parallel C_3) \sqsubseteq (C_1; C_2) \parallel C_3$ can be extended to the analogue of the distributive law proven by Lemma 4.4.13.

The validation of refinement methods in this thesis suggest that laws that do not exploit properties of the state are not sufficiently powerful to justify some useful refine-

ments that one would reasonably expect to hold. Our generic framework for refinement suggests that this insufficiency is due to the fact that transition traces (as well as stateless simulation) allow arbitrary interference to occur during a computation. It is to be expected that the approach based on transition traces can be used to derive more powerful laws by limiting the possible interferences. One attempt at limiting the interference in a transition trace based method is described by Dingel in [48] where he investigates how the interferences allowed by transition traces can be linked to the context of a program. In this thesis, the idea of relating the interference of a program to its context led to the development of convex refinement.

It would be interesting to develop a theory of transition traces which is parameterized by the interference. This could yield a generic framework, analogous to the one based on simulation described in Chapter 5, which allows a more flexible way of formalizing assumptions about the possible interferences.

The general applicability of the transition traces approach to concurrent languages that communicate through asynchronous communication is shown by De Boer et al. in [18]. There it is compared with the failure semantics which is used for assigning meaning to languages based on synchronous communication.

A transition traces approach for Gamma has been developed by Sands in [105].

An interesting feature that is laid bare by the simulation approach is the distinction between strong and weak refinements. By their construction, transition traces are closed under "stuttering". As a result they seem insensitive to this difference.

An interesting alternative for using a multi-step transition system for modelling the concurrent execution of rewrite rule is the pomset model [100]. We chose to base our method of refinement on bisimulation because it has a powerful proof method associated with it.

8.2.3 Temporal Logic

Temporal logics (e.g. [97], [86]) use properties of sequences of states or sequences of actions for specifying and reasoning about parallel systems.

Because of its non-operational style of reasoning, a temporal logic is used in the UNITY [23] approach. programs based on temporal logic is illustrated by Singh in [110]. There Singh presents a refinement rule which justifies the strengthening of guards of UNITY statements. This rule closely resembles the convex strengthening law that we prove for our coordination language by Lemma 6.2.1.

The applicability of temporal logic to Gamma is illustrated by Reynolds in [103] where he uses it for defining a semantics for Gamma. It would be interesting to extend this approach to schedules and investigate whether it could serve as the basis of a method of refinement of schedules. Since temporal logics are well suited for dealing with reactive processes, such a method could complement the more output-oriented approach followed in this thesis.

8.2.4 Algebraic Methods

There is a large body of work on algebraic methods for the description of communicating processes (most notably ACP [11], CCS [90], CSP [76]). The algebraic approaches towards reasoning about parallel systems consists of using variables (and constants) to denote events and using operators, such as sequential and parallel composition, as combinators for processes. According to this approach, properties of processes can be described in an algebraic framework.

In this area of research, the main focus is on processes that have private state (and communicate through message passing). In order to obtain a decoupling between computation and communication we used a programming model which employs asynchronous communication via a shared data-space (i.e. shared state). As a consequence of this difference, the behaviour of scheduled Gamma programs is defined in terms of configurations $\langle s, M\rangle$ which consist of a schedule component (a process part) and a multiset component (a state part).

This difference from the private-state approaches prohibits that (bi)simulation-based notions of equivalence yield a (pre)congruence relation over schedules. Hence, it complicates the construction of a method for algebraic reasoning about shared-memory processes. However, in Chapter 5 we have shown how (pre)congruent refinement (equivalence) relations can be obtained. We will discuss some alternative approaches for solving this problem that have been proposed in the literature.

In [63] Groote and Ponse first consider a language which contains combinators for prefixing, nondeterministic choice and guards. They employ a two-stage construction to obtain a congruence for this language. Firstly, an equivalence relation over process-state configurations is defined using bisimulation . Next, an equivalence over programs is defined which equates two processes if they are bisimilar for all possible initial states. The generic framework of refinement from Chapter 5 suggests that the quantification of the state-component in the notion of equivalence can be interpreted as taking all possible

interference into account. From this point of view, the approach of Groote and Ponse defines a notion which allows interference only before the first action, and no interference from the the first action onwards. This approach seems prompted by the fact that it technically (from a mathematical point of view) solves the problem, however it does not reflect a sensible assumption about interference in a shared memory setting.

Next, the language considered in [63] is extended with operators for parallel composition. The notion of equality of processes sketched above does not yield a precongruence in this setting. This is solved by adapting the notion of bisimulation such that it allows interference at every stage of execution.

This latter approach is essentially the same as that followed in [34] where Ciancarini, Gorrieri & Zavattaro develop a bisimulation-based equivalence for Gamma programs (based on a Plotkin-style operational semantics [96]) by universally quantifying over the multisets.

The universal quantification of the state-component of these notions of bisimulation corresponds to the type of quantification used to define stateless refinement. Hence, these notions do not support refinements that depend on properties of the state. Based on our experience with notions of refinement in this thesis, we expect that these stateless-like notions are not (by themselves) sufficiently powerful to justify all refinements that one would reasonably expect to hold (and hence expect to be able to use) in a shared memory setting.

An alternative approach is obtained by avoiding the distinction between operations and data by considering both of these as elements of the same algebra. This essentially boils down to considering a datum as a process that can offer its value and may engage in a communication with processes that are looking to use this datum. This idea has been used by Ciancarini et al. in [36] and by De Nicola and Pugliese in [44, 45] for giving semantics for Linda (see [22]).

These approaches manipulate a mathematical structure which includes both the data structure and control structure. In our opinion, this obfuscates the coherence of the coordination structure. As a result it is not clear how the coordination structure can be refined in a modular way. Also, it is unclear how properties of the data can be used in refinements.

Besides the best established algebraic approaches, a noteworthy effort is that of the "Calculus of Broadcasting Systems" [98, 99] - CBS for short. CBS is a CCS-like calculus which has broadcasting rather than hand-shaking as basic primitive for communication. In [107] a translation of a class of Gamma programs into CBS is examined.

As a consequence of choosing broadcasting as communication primitive, there are a number of similarities with the programming model we used in this thesis. In CBS holds that once a datum has been broadcast, it is available to all "listeners". Similarly, we have in Gamma that once a datum has been inserted in the multiset it is available to all rewrite rules that wish to use it. Related to this is the fact that a listener in CBS and a rewrite rule in Gamma do not have to identify the source (broadcasting process or creating rewrite rule) of a datum in order to use it. Hence, analogous to the Gamma model, the broadcast communication abstracts from locality.

However, in contrast to the shared state of the Gamma model, CBS has processes which have private state. A further difference is that it is inherent in the principle of broadcasting that only one single process may broadcast at a time. Thus only one datum is available for all actions that are looking to engage in a communication. This limits the number of computations that can be performed at any time in a way that is not inherent to the logic of the problem.

In the setting of process algebras with private state and message-passing there have been some efforts towards a generic theory of equivalence aimed at obtaining precongruences over process-terms [64], [17] and [46]. Just as our approach, these are based on the idea of using (bi)simulation as notion of refinement (equivalence) over behaviours. In contrast to our approach, these efforts have fixed the notion of equivalence (as bisimulation) and predict that it is a (pre)congruence if the semantic rules that define the structural operational semantics fit certain formats. Therefore, these approaches are more oriented towards investigating the design of (primitives for) programming languages, rather than investigating a generic framework of notions of refinement.

8.3 Concluding Remarks

> *[...] one of the outstanding challenges in concurrency is to find the right marriage between logical and behavioural approaches.*
> Robin Milner, p. 4, [90]

During the stages of our method for program design, we separately focus on computation (functionality) and coordination (behaviour). These phases lend themselves for different methods of reasoning.

We can adequately reason about the correctness of a computation by means of a logic which abstracts from behaviour. Subsequently, during the design of a coordination

strategy emphasis is on the behavioural aspects of a system and this is reflected by the methods which we have proposed in support of that stage.

It is at the behavioural level that the additional complexity of reasoning about parallel systems manifests itself. This complexity stems from the interaction of behaviour of individual components. All of the methods for reasoning about parallel systems deal with interaction between individual components in one way or another. For instance by excluding interference in time (critical sections) or space (by ensuring that control over a variables is local). In our framework interaction plays a central rôle at the level of behaviour by using the degree of interference as a parameter in our theory of refinement.

9 Concluding Remarks

In this thesis we presented a formal methodology for the design of parallel and distributed programs based on the separation of computation and coordination. In this chapter we will describe the contributions of this work, reflect on some issues that arose during the research for this thesis, and mention some directions in which it would be interesting to pursue further research.

9.1 Contributions of this Thesis

In [58] Gelernter and Carriero formulate the thesis that programming models can be viewed as consisting of a computation component and a coordination component and advocate the advantages of addressing these issues using separate languages.

Their approach to parallel program design [21] consists of first selecting one of several paradigms of parallel computation. Secondly, this paradigm is expressed using a combination of a (sequential) computational base-language (such as C) and the Linda set of primitives for asynchronous communication through a shared data space. Although they separate computation and communication, their approach merges computation and coordination aspects into a single program text. Furthermore the transition from specification to program is not supported by formal methods of reasoning.

In his thesis [79], De Jong expresses his ideas on how a computational model based on multiset transformations and a separate coordination model based on scheduling can be integrated in a formal method of parallel program design.

Building on these insights, we present in this thesis the first programming methodology that formally supports the separate design of computation and coordination aspects. To this end, we developed a body of formal methods which comprises the following contributions.

- We proposed a formal semantics for the multiset programming model Gamma. This semantics differs from other semantics for Gamma in that it explicitly models

the parallelism of Gamma programs.

- We proposed a coordination language to be used in combination with the Gamma model and formally defined its syntax and semantics. The coordination language provides a framework which highlights the differences and commonalities between control strategies. As such, it provides a suitable mechanism for the investigation of control strategies.

- We devised a method which shows how to construct, for a given Gamma program, a coordination strategy that describes the most general behaviour of that program. The adequacy of this construction was formally proven.

- In support of a method for the step-wise derivation of coordination strategies, we developed a generic theory of refinement, based on the notion of simulation, for parallel processes that operate on shared memory.

  As is characteristic for formal methods for reasoning about parallel processes, the notion of interference plays a central rôle in our theory of refinement. The theory we developed has a parameter which can be set to reflect different assumptions about interference and thereby induces a space of refinement notions. We studied the effect of different choices of the interference parameter on properties of the corresponding refinement relation. This showed that the interference parameter induces a partial ordering on this space of refinement relations. Furthermore, we identified conditions on the interference parameter that are sufficient to ensure that the corresponding notion of refinement is a precongruence.

  Special attention is paid to a triplet of notions of refinement: statebased, stateless and convex refinement. For each of these notions we suggested methods that can be used in proving refinements.

  - Statebased refinement is the most powerful notion. A statebased refinement can be proven by constructing a statebased simulation. The practical use of plain statebased refinement is limited by some drawbacks. We list these drawbacks together with the methods we have proposed for alleviating them.

    * Proving a refinement using a simulation relation requires a way of dealing with the combinatorial explosion that is due to the large number of possible ways of performing computations. In Section 3.2.2 we present

a method which reduces reasoning about parallel behaviour to reasoning about the corresponding interleaved (sequential) behaviours.

* Statebased refinement is not a precongruence, which prohibits the use of the (in)equations it induces in a modular equational style. This is generally considered to limit practical applicability. However, in Section 7.5 we observed that some contexts offer possibilities for using statebased refinement in a modular fashion while incurring only a limited penalty in terms of additional proof obligations.

A further aspect in favour of statebased simulation is that due to the correspondence with correctness proofs based on assertional proof methods (such as the weakest precondition calculus), statebased simulation should be straightforward to use for anyone familiar with the former method.

– The stateless refinement relation is a precongruence. Consequently, we may employ the powerful proof method of algebraic reasoning using the stateless refinements. However, because stateless refinement makes worst-case assumptions about interference, it justifies fewer refinements than the other notions of refinement.

– Convex refinement improves on both statebased and stateless refinement in that it allows the use of properties of the multiset for proving refinements (as statebased) while also being a precongruence (as stateless).

The properties of the multiset that are used by convex and statebased refinements are often properties that were already proven in order to establish the correctness of the associated Gamma program. Hence, the correctness proof of a Gamma program can be reused in the development of coordination strategies.

Convex refinement formalizes a notion of robustness/fault tolerance. Given the possible interferences from the environment, it distinguishes algorithms that produce correct output from those that do not.

The step-wise derivation of the coordination component reveals the decisions that underlie certain execution strategies. Different design decisions disclose how variations on strategies form a family of algorithms that are formally related by refinement.

In Chapter 7 we present a number of case studies which show that the computation

and coordination aspects of program can be developed separately and formally using the methods presented in earlier chapters. Furthermore, these case studies give insight into the strengths and weaknesses of the different techniques for proving refinements. More detailed conclusions on the proof methods and their rôle in the derivation of coordination can be found in Section 7.6.

9.2 On what we have rejected

In the preceding chapters we have presented the product of our research. This does not completely cover the decisions that were made in order to arrive at this product. In the next sections we will discuss some of these topics and we will indicate the motivations for not including them in the core methods.

9.2.1 Nondeterministic Choice

Process algebras such as CCS [90], CSP [76], ACP [11], have in common with our coordination language that they constitute formal languages that can be used to define behaviours in terms of some collection of basic actions. All of these process algebras include a combinator for nondeterministic choice and it was suggested in [112] that we include such a combinator in our language.

We have decided not to do so because of the following reasons"

- Firstly, the coordination language was designed primarily for modelling the behaviour of Gamma programs. The aim was to accomplish this with as small a language as was possible. From the preceding chapters can be seen that this goal was realized without the need for nondeterministic choice. The reason that we had no need for a nondeterministic choice has to do with the fact that the purpose of our coordination language differs from that of process algebras.

  - Process algebras take a descriptive point of view where process terms are used to describe the observable behaviour of a system. In such cases the internal structure of (or causal relations within) a system are hidden (possibly by an explicit hiding mechanism) from the observer. This absence of knowledge may cause the behaviour to appear nondeterministic. Hence, from this point of view, a nondeterministic operator is necessary for describing behaviour of systems.

  - In contrast, we use our coordination language from a prescriptive point of view to dictate the behaviour of a system. Hence, we use it for defining the actual structure and causal relationships of a process. From this point of view the nondeterminism of the parallel combinator suffices for describing that we do not care in which order subprocesses are executed.

    This prescriptive point of view also eliminates the need for a hiding operator which is usually the cause of the observation of nondeterministic behaviour.

- A combinator for nondeterministic choice has a global character. Consider the nondeterministic composition of a number of schedules, where each schedule can be executed on a different machine (hence a different location): $s_1 \oplus \ldots \oplus s_n$. The semantics of nondeterministic choice prescribes that one schedule is allowed to engage in a transition and the other schedules are terminated (pre-empted). This suggests that all processes must have knowledge of which schedule is the "chosen one" at the same moment during execution. Clearly, this requires a global synchronization.

  In the setting of parallel and distributed system, optimizing the locality of computations minimizes the need for synchronization. Therefore, "locality" is an important principle that underlies the philosophy behind Gamma. We wanted to accord with this principle in the design of the coordination language.

An additional advantage of the absence of nondeterministic choice in our coordination language is that we do not have complications in showing (pre)congruence of weak notions of refinement such as are experienced by Milner for CCS.

In Section A.3 we investigate the extension of the coordination language with an operator $\oplus$, for nondeterministic choice.

9.2.2 Synchronous Parallel Composition

During the design of the coordination language we have contemplated the incorporation of a combinator for strictly synchronous or "must" parallel composition.

The combinator, denoted "Ξ", would range over multi-sets of rewrite rules. The special case of the binary synchronous parallel combinator is denoted "$\mid$". An attempt

at defining its semantics is as follows

$$\frac{\langle r_i, M\rangle \xrightarrow{\lambda_i} \langle \mathsf{skip}, M_i\rangle \quad \forall i : 1 \leq i \leq n}{\langle \Xi_{i=1}^{n} r_i, M\rangle \xrightarrow{\lambda} \langle \mathsf{skip}, M'\rangle} \quad \text{where} \quad \begin{array}{l} \forall i : M \models \lambda_i \bowtie \Sigma_{j\neq i}\lambda_i \\ \lambda = \lambda_1 \cdot \ldots \cdot \lambda_n \end{array}$$

Formally, the definitions of $M \models$ and $\bowtie$ should be extended to deal with ε-transitions in the obvious way. The semantic rule for Ξ should be read as follows: a synchronous composition can make a transition only if all rewrite rules can perform individual transformations that are independent. In that case all these rewrites must take place in a single transition.

Some considerations in favour of using a combinator for "must" parallelism are the following.

- A combinator which prescribes "must" parallelism rather than the "may" parallelism that we use throughout this thesis, would seem to allow a more specific description of behaviour. Binary "may" parallel composition of rewrite rules allows three orders of execution, while "must" parallelism allows only one.

  This can be illustrated using the combinator for nondeterministic choice introduced in the previous section. The "may" parallel composition of rewrite rules can be described as a nondeterministic choice over sequential composition and "must" parallel composition in the following manner (similar to ACP):

  $$r_1; r_2 \oplus r_2; r_1 \oplus r_1 \mid r_2 \;\simeq\; r_1 \parallel r_2$$

  Hence, by the laws for nondeterministic choice, we would get that $r_1 \mid r_2 \leqslant r_1 \parallel r_2$.

  These properties suggest that synchronous parallel composition is a more primitive combinator than "may" parallel composition.

The reasons we decided against this operator are the following:

- The "must" parallel combinator is not needed to define the most general schedule, nor does it play a crucial role in the derivation process. Hence, it would violate our goal to keep the coordination language as simple as possible.

- The semantics of "must" parallel requires global synchronization of the cooperating rewrite rules. While this is natural in SIMD systems, we did not want to make it primitive to our language.

Instead of having a combinator for "must" parallel composition, we suggest that noninterference of rules or schedules may be annotated in the schedule text.

9.2.3 Single Step Semantics

A structural operational semantics for Gamma was first put forward in [65, 66]. It describes a mapping from programs to transition systems in such a way that every individual transition models the execution of exactly a single rewrite rule from a Gamma program. In this section we show that this does not model the parallelism of Gamma programs in a way that can be exploited using simulation as the notion of refinement.

Since the schedule language is intended to steer the behaviours of Gamma programs, it cannot describe behaviour that is not already possible for a Gamma program. If the semantics for Gamma programs requires that every transition corresponds to the execution of a single rewrite rule, then the same must be the case for the semantics of schedules. As a consequence, the notions of simulation on this semantics equates parallel and sequential composition of rewrite rules.

To show this formally, we consider the single step semantics (i.e. the semantic rules from Figure 3.3 without (N3) and (N4) that are responsible for multi-step transitions).

Lemma 9.2.1 $r; r \simeq r \parallel r$

Proof The result follows by showing $r; r \leqslant r \parallel r$ and $r \parallel r \leqslant r; r$. We consider these cases in turn.

- $r; r \leqslant r \parallel r$: Follows straightforwardly by showing that $\{(r; r, r \parallel r), (r, r), (\mathsf{skip}, \mathsf{skip})\}$ is a strong stateless simulation.

- $r \parallel r \leqslant r; r$: Let $\mathcal{R} = \{(r \parallel r, r; r), (r, r), (\mathsf{skip}, \mathsf{skip})\}$. We show that $\mathcal{R}$ is a strong stateless simulation. First consider the pair $(r \parallel r, r; r)$.
  transition
  Here, it is important to realise that (N3) and (N4) may not be used to derive a transition for $r \parallel r$. Hence, the transition that we have to consider for $r \parallel r$ are derived by (N2). By symmetry, there is only one transition to consider.

  - $\langle r, M\rangle \xrightarrow{\lambda} \langle \mathsf{skip}, M'\rangle$. Then by (N5) $\langle r; r, M\rangle \xrightarrow{\lambda} \langle r, M'\rangle$. Clearly $(r, r) \in \mathcal{R}$.

  termination: Straightforward.

It is straightforward to show that the remaining pairs (r, r) and $(\mathsf{skip}, \mathsf{skip})$ satisfy the definition of strong stateless simulation.

□

9.3 Future Work

In this section we sketch some ideas for future work.

9.3.1 Data Structures and Data Refinement

In the Gamma model both actions and data are unordered. In this thesis we have chosen to order the actions to control the behaviour of programs. It would be interesting to investigate whether a supplementary (or complementary) approach could consist of increasing the ordering on data.

Using such an approach, the data structure would guide the manner in which the program is executed. The first question that arises when pursuing this approach is "What is the best way to define a data structure without invalidating the basic qualities of the programming model?"

There have been some approaches towards super-positioning a data structure on Gamma programs by imposing some type discipline:

- In [89] McEvoy presents a more pragmatic view on types called "chromatic typing". There, he builds a type system for tuples out of two components:
  - An "underlying" type which denotes the kind of value that a tuple represents; this is built out of primitives such as types for atomic data items such as `num`, `bool` and `string`.
  - A "chromatic type" which is used to indicate which rewrite rules may be applied to a tuple. This can be used to keep track of whether there is data available for a rewrite rule to operate upon.

  He claims that "Chromatic types allows the partitioning of the multiset in a number of typed sets ... in a way which reduces the overhead of termination detection, and which complements the 'spirit' of Gamma functions as multiset transformers".

- In [53] and [55] Fradet and Le Metayer define "Structured Gamma". This is an extension of the Gamma model with "Shape Types" as described in [54] This theory associates an address with every element of the multiset and defines data structures in terms of relation over these addresses. These addresses may be used in the reaction conditions of rewrite rules and may be transformed by an actual rewriting.

  These papers describe how a number of data structures can be expressed as shape types and present a type checking algorithms that can be used to verify that a Gamma program maintains the data structure.

Analogous to our approach for the coordination language, it would be interesting to investigate an algebraic approach towards the definition of data structures. A potential point of departure could be the Boom Hierarchy of relations as it is used for the derivation of algorithms in [77]. The hierarchy is based on the observation that data types can be described by a combinator of terms which satisfies certain algebraic properties. If a combinator enjoys more algebraic properties, then the corresponding data structure is more flexible.

To illustrate, consider a combinator (of terms) that has no additional structure. Any term formed using this combinator corresponds to a tree (i.e. its parse tree). By adding associativity, the order of composition is lost which equates terms as if they were lists (or sequences). Adding commutativity on top of this, eliminates the order of the elements in lists. The resulting structure corresponds to that of multisets. Furthermore, if idempotency is added as property of the operator, then a term becomes indifferent to multiple occurrences. This reduces multisets to sets.

Consider an algebra which contains all of the aforementioned combinators. Then two elements are neighbours in a term from this algebra if this term can be rewritten (using the algebraic properties that it satisfies) into a term where the elements are syntactically adjacent. This induces equivalence classes of neighbours for all elements in the multiset (and on top of this a notion of distance).

The notions of neighbourhood can be used to require that the execution mechanism first selects all immediate neighbours. If that does not yield a match, then the selection process proceeds with selecting elements that are at a distance one larger than in the previous phase. The choice between elements at equal distance may still be made nondeterministically.

Our point of departure would, in general, be the multiset (or the set in special cases).

This would be represented by a term where all tuples are composed using the multiset combinator. Replacing the multiset combinators with combinators that have fewer algebraic properties effectively reduces the neighbourhoods of tuples, hence provides stronger guidance for the order of selecting data.

This approach seems suitable for the step-wise development of data structures because it allows hybrid (some parts list, some part multiset) representations of the data structure. However it also raises some questions: At what location in the data structure should elements created by a rewrite rule be inserted? And how does this relate to where the search continues? Is it possible to transfer to current location of search to another location? Is this approach sufficiently expressive to define arbitrary data structures?

As final point of further investigation, we mention an approach which includes simulations. In defining the simulation relations used in this thesis care has been takes to phrase them in sufficiently general terms to allow a change of data representation using a structure preserving mapping. We illustrate this idea by the following example.

Example 9.3.1 *The summation program* $add \mathrel{\widehat{=}} x, y \mapsto x + y$ *from Section 7.1 may be applied to a multiset* $M = \{a_1, \ldots, a_n\}$ *which contains a collection of numbers. A more concrete representation can be obtained by representing the multiset by means of tuples which associate a value with a position in the sequence; e.g.* $M^c = \{(i, a_i) \mid a_i \in M\}$.

The program needs to be modified accordingly and needs to take the different data representation in account. We consider the following program $add^c \mathrel{\widehat{=}} (i, x), (j, y) \mapsto (i, x + y)$. *Now let* $\delta : \mathbb{M} \to \mathbb{M}$ *be an abstraction mapping defined by* $\delta(M) = \{x \mid (i, x) \in M\}$. *The definitions of simulation can be extended to include the correspondence between an abstract and concrete data-representation by the following adjustment:*

Definition 9.3.2 *A binary relation on configurations* $\mathcal{R} \subseteq \mathbb{C} \times \mathbb{C}$ *is a* strong δ simulation *if* $(\langle s, M\rangle, \langle t, N\rangle) \in \mathcal{R}$ *implies, for all* λ,

1. $N = \delta(M)$
2. $\langle s, M\rangle \xrightarrow{\lambda} \langle s', M'\rangle \Rightarrow \exists t' : \langle t, N\rangle \xrightarrow{\delta(\lambda)} \langle t', N'\rangle$ *such that* $(\langle s', M'\rangle, \langle t', N'\rangle) \in \mathcal{R}$
3. $s \equiv \mathsf{skip} \;\Rightarrow t \equiv \mathsf{skip}$

In this example it is interesting to note that once a data representation has been chosen, a more specific behaviour has to be specified by the Gamma program. In this case the positions that indicate the elements that are removed and the position where their sum is inserted may be used to suggest an order of computation, as is illustrated by the following alternatives:

- $add_1 \mathrel{\widehat{=}} (i, x), (j, y) \mapsto (min(i, j), x + y)$
- For an alternative initialization $M' = \{(n + i - 1, a_i) \mid 1 \leq i \leq n\}$, the program $add_2 \mathrel{\widehat{=}} (2i, x), (2i + 1, y) \mapsto (i, x + y)$ already induces a recursive doubling style of computation (for $n = 2^k$ for some $k \geq 0$).

Some aspects by which a mechanism for data structuring should be judged are the following:

- the opportunities it provides for (step-wise) refinement
- its ability to reason about the space-usage
- its compatibility with refinement of the control structure

Furthermore, it would be interesting to find out if some notion of accordance between a data structure and a control structure could be defined which can be used to indicate the time complexity of their combined use.

9.3.2 Schedules for Tropes

In [67] Hankin, Le Métayer and Sands proposed a quintet of general rewrite rules: Transmuter, Reducer, OPtimizer, Expander and Selector. They argue that these rules captured most common patterns of computation and could be used to define any program. The advantages behind this idea are twofold:

- A logic for reasoning about programs can be tailored to a specific set of computational primitives.
- Properties of a primitive can be exploited to devise an efficient implementation for it. If the given set of primitives can be implemented efficiently, then any program expressed in terms of these primitives can be implemented efficiently.

The approach described in this thesis can be used to derive coordination strategies for the TROPES templates. Consider, for example, the reducer primitive.

$$R(C, f) = x, y \mapsto f(x, y) \Leftarrow C(x, y)$$

The summation program from Section 7.1 is an example of a reducer. The schedules we present there could also be derived using the above definition of a reducer rule.

Parameterized by a rewrite rule r, the recursive doubling strategy would look as follows

$$RD_r(i) \mathrel{\widehat{=}} (i > 1) \triangleright (r^{\lfloor i/2 \rfloor}; r^{\lceil i/2 \rceil})$$

Now, $RD_{R(true,*)}(i)$ yields a recursive doubling schedule for the reducer rule $R(true, *)$. Hence, $\langle RD_{R(true,*)}, \{2, \ldots, n\}\rangle$ denotes a parallel strategy for computing $n!$ which would not readily be obtained using other programming paradigms.

Following this approach, one could have a library of coordination strategies based on particular properties of the TROPES and possibly some parameters of machine architectures. A program could then be composed by expressing a program in terms of TROPES and selecting suitable coordination strategies.

An area of research that is closely related to this approach is that of *skeletons*. A skeleton is a template of a high-level control strategy that is parameterized by the basic actions that need to be performed. These templates can be instantiated by supplying a suitable set of basic actions. An important motivation for the development of skeletons is that they can be used to hide the actual implementation details from the programmer, while still providing the opportunity for exploiting parallel architectures.

Using our approach sketched above, programmers benefit from skeletons at a different level: they would be spared the details of refinement proofs, while still obtaining coordination strategies that are tailored to their needs.

It would be interesting to find out whether the separate specification of coordination strategies could contribute to solving the problem of composing skeletons.

9.3.3 Automated Support

The fallibility of humans does not stop at writing programs, they also make errors in proofs. Therefore, any formal method should be supported by automated tools. These tools can verify and assist in refinement proofs or even steer the derivation by suggesting which refinement ways are applicable.

For proofs by simulation there may be opportunities to apply techniques from the expanding field of model checking [38]. Algebraic proofs may be supported by general purpose theorem-provers.

An automated tool for the derivation of coordination strategies typically requires heuristics to guide its search. A good candidate for such a heuristic would be the expected performance that could be obtained by a schedule (given some machine-architecture).

A Definition of Basic Concepts

A.1 Congruence

Definition A.1.1 *A relation $\mathcal{R}$ over a term-algebra Σ is a congruence iff*

1. *$\mathcal{R}$ is a equivalence relation on Σ,*
2. *for all f, for all $\overline{s}, \overline{s}' \in \Sigma$, if $\overline{s}\mathcal{R}\overline{s}'$, then $f(\overline{s})\mathcal{R}f(\overline{s}')$.*

Definition A.1.2 *A relation $\mathcal{R}$ over a term-algebra Σ is a precongruence iff*

1. *$\mathcal{R}$ is a partial order on Σ,*
2. *for all f, for all $\overline{s}, \overline{s}' \in \Sigma$, if $\overline{s}\mathcal{R}\overline{s}'$, then $f(\overline{s})\mathcal{R}f(\overline{s}')$.*

A.2 On Multisets

Definition A.2.1 *Let A be a set.*

1. *A multiset over A is a function $M : A \to \mathbb{N}$.*
2. *Let $\mathbb{M}$ be the set of multisets; i.e. $\mathbb{M} = \{M \mid M \text{ is a multiset}\}$.*

Definition A.2.2 *Let A be a set and let M and N be multisets over A.*

1. *a is a member of M, denoted $a \in M$, if $M(a) > 0$.*
2. *M is equal to N, denoted $M = N$, if $M(a) = N(a)$ for all $a \in A$.*
3. *M is a sub-multiset of N, denoted $M \subseteq N$, if $M(a) \leq N(a)$ for all $a \in A$.*

Definition A.2.3 *Let A be a set and let M and N be multisets over A.*

1. *$M \cup N = \{(a, M(a) + N(a)) \mid a \in A\}$ is the* union *of M and N.*

2. $M \cap N = \{(a, min(M(a), N(a))) \mid a \in A\}$ *is the* intersection *of* M *and* N.

3. $M \ominus N = \{(a, M(a) \dot{-} N(a))) \mid a \in A\}$ *where* $x \dot{-} y = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$ *is the* difference *between* M *and* N.

Definition A.2.4

1. *Let* N *and* N' *be multisets. A* multiset substitution *that replaces* N *by* N' *is a function* $\sigma : \mathbb{M} \to \mathbb{M}$ *that is written as* N'/N. *Formally*

$$\sigma(M) = \begin{cases} (M \ominus N) \cup N' & \text{if } N \subseteq M \\ M & \text{otherwise} \end{cases}$$

   To conform with conventional notation for substitution, we also write $M[\sigma]$ *to denote the application of* σ *to* M.

2. *The special symbol* ε *is used to label transitions that do not affect the multiset. Formally, it can be defined as the identity function on multisets:* $\varepsilon(M) = M$.

   This makes ε *an identity for composition of multiset substitutions; i.e.* $\varepsilon \cdot \sigma = \sigma = \sigma \cdot \varepsilon$.

Definition A.2.5 *Let* M *be a multiset and let* $\sigma_1 = N_1/M_1$ *and* $\sigma_2 = N_2/M_2$ *be multisets substitutions.*

1. σ_1 *is* independent from σ_2 *in* M, *denoted* $M \models \sigma_1 \triangleleft \sigma_2$, *if* $N_1 \subseteq (M \ominus N_2) \cup M_2$.

2. *If* σ_1 *and* σ_2 *are mutually independent from each other, more succinctly called* independent, *then we write* $M \models \sigma_1 \bowtie \sigma_2$.

Lemma A.2.6 *Let* M *be a multiset and let* $\sigma_1 = N_1'/N_1$ *and* $\sigma_2 = N_2'/N_2$ *be multiset substitutions. If* $N_1 \subseteq M$, $N_2 \subseteq M$ *and* $M \models \sigma_1 \bowtie \sigma_2$, *then* $\sigma_2 \cdot \sigma_1(M) = \sigma_1 \cdot \sigma_2(M)$.

Proof Recall that $M(x)$ denotes the number of elements x in multiset M. We reason

as follows

$$
\begin{array}{ll}
& x \in \sigma_2 \cdot \sigma_1(M) \\
\Leftrightarrow & \qquad\qquad \text{subst. } \sigma_1, N_1 \subseteq M \\
& x \in \sigma_2((M \ominus N_1) \cup N_1') \\
\Leftrightarrow & \qquad\qquad \text{subst. } \sigma_2, N_2 \subseteq (M \ominus N_1) \cup N_1' \\
& x \in (((M \ominus N_1) \cup N_1') \ominus N_2) \cup N_2' \\
\Leftrightarrow & \qquad\qquad \text{def. } \ominus, \cup, N_1 \subseteq M, N_2 \subseteq M \ominus N_1 \cup N_1' \\
& M(x) - N_1(x) + N_1'(x) - N_2(x) + N_2'(x) \geq 1 \\
\Leftrightarrow & \qquad\qquad \text{arithmetic} \\
& M(x) - N_2(x) + N_2'(x) - N_1(x) + N_1'(x) \geq 1 \\
\Leftrightarrow & \qquad\qquad \text{def. } \ominus, \cup, N_2 \subseteq M, N_1 \subseteq M \ominus N_2 \cup N_2' \\
& x \in (((M \ominus N_2) \cup N_2') \ominus N_1) \cup N_1' \\
\Leftrightarrow & \qquad\qquad \text{subst. } \sigma_2, N_1 \subseteq (M \ominus N_2) \cup N_2' \\
& x \in \sigma_1((M \ominus N_2) \cup N_2') \\
\Leftrightarrow & \qquad\qquad \text{subst. } \sigma_1, N_2 \subseteq M \\
& x \in \sigma_1 \cdot \sigma_2(M)
\end{array}
$$

□

A.3 Pre-emptive Nondeterministic Choice

In this section, we sketch the results of investigating the extension of the coordination language with an operator $\oplus$, for nondeterministic choice.

Let us extend the operational semantics with an inference rule $(N \oplus)$ which defines the behaviour of the nondeterministic choice combinator.

$$(N \oplus) \quad \frac{\langle s_1, M\rangle \xrightarrow{\lambda} \langle s_1', M'\rangle}{\langle s_1 \oplus s_2, M\rangle \xrightarrow{\lambda} \langle s_1', M'\rangle} \tag{A.1}$$

Figure A.1: Semantic Rule for Nondeterministic Choice

Additionally, we capture the commutativity of nondeterministic choice by defining the structural congruence

$$s_1 \oplus s_2 \equiv s_2 \oplus s_1$$

We write $\Sigma_{i=1}^{n} s_i$ to denote the nondeterministic choice over s_i for $i : 1 \leq i \leq n$.

We show that strong stateless, and consequently strong convex, refinement are preserved by nondeterministic composition. Hence, the corresponding refinement notions are precongruences for the coordination language extended with nondeterministic choice.

Lemma A.3.1 *If $s_1 \leqslant t_1$ and $s_2 \leqslant t_2$, then $s_1 \oplus s_2 \leqslant t_1 \oplus t_2$.*

Proof Let $\mathcal{R} = \{(s_1 \oplus s_2, t_1 \oplus t_2) \mid s_1 \leqslant t_1, s_2 \leqslant t_2\}$. The result follows by straightforwardly shoowing that $\mathcal{R}$ is a strong stateless simulation. □

It is straightforward to prove the following laws for nondeterministic choice. which show that $(\oplus, \mathsf{skip})$ is a commutative monoid.

$$\begin{array}{lcl} \mathsf{skip} \oplus s & \simeq & s \\ s_1 \oplus s_2 & \simeq & s_2 \oplus s_1 \\ (s_1 \oplus s_2) \oplus s_3 & \simeq & s_1 \oplus (s_2 \oplus s_3) \end{array}$$

Another straightforward result is the idempotency of $\oplus$:

$$s \simeq s \oplus s$$

Furthermore, we can show how nondeterministic choice relates to the other operators from the coordination language.

$$\begin{array}{lclclclcl} r \to s[t] & \oplus & r \to s'[t] & \oplus & r \to s[t'] & \oplus & r \to s'[t'] & \leqslant & r \to (s \oplus s')[t \oplus t'] \\ c \triangleright s[t] & \oplus & c \triangleright s'[t] & \oplus & c \triangleright s[t'] & \oplus & c \triangleright s'[t'] & \simeq & c \triangleright (s \oplus s')[t \oplus t'] \\ s_1 \parallel s_3 & \oplus & s_1 \parallel s_4 & \oplus & s_2 \parallel s_3 & \oplus & s_2 \parallel s_4 & \leqslant & (s_1 \oplus s_2) \parallel (s_3 \oplus s_4) \\ s_1; s_3 & \oplus & s_1; s_4 & \oplus & s_2; s_3 & \oplus & s_2; s_4 & \leqslant & (s_1 \oplus s_2); (s_3 \oplus s_4) \end{array}$$

The last law is the classic example of an equivalence that does not hold in bisimulation-based approaches, but does hold in trace-based approaches for reasoning about nondeterministic processes. This law shows, however, that a refinement holds in one direction. (A more common, but less general, formulation of this law is obtained by taking s_1 and s_2 to be equal: $(s_1; s_3) \oplus (s_1; s_4) \leqslant s_1; (s_3 \oplus s_4)$.)

Using these laws for distribution, the nondeterministic choices of a schedule can always be moved to become the most outward operator of a schedule (and thus yield an expansion law analogous to Milner [90]).

The kind of refinement made possible by nondeterministic choice is illustrated by the following law

$$s_1 \leqslant s_1 \oplus s_2$$

Hence, in combination with the distribution laws, refinement laws using nondeterministic choice consist of selecting one of a number of possible disjunct alternatives.

To illuminate the relation between replication and nondeterministic choice, Lemma A.3.2 confirms the intuition that $!s$ stands for an arbitrary number of instances of s executing in parallel.

Lemma A.3.2 *For all* $k : k \geq 1 : \Sigma_{i=1}^{k} s^i \leqslant !s$

Proof By induction on k:

- $k = 1$: $s \leqslant !s$.
- $k > 1$:

$$\begin{array}{lll} & \Sigma_{i=1}^{k+1} s^i & \\ \simeq & & \text{def. } \Sigma \\ & (\Sigma_{i=1}^{k} s^i) \oplus s^{k+1} & \\ \leqslant & & \text{induction hypothesis} \\ & !s \oplus s^{k+1} & \\ \leqslant & & \text{for all } n \geq 1 : s^n \leqslant !s \\ & !s \oplus !s & \\ \simeq & & \text{idempotency } \oplus \\ & !s & \end{array}$$

□

B Glossary of Notation

Basic Concepts

$\mathbb{M}$	set of multisets M, N
$\mathbb{P}$	set of Gamma programs P, R
$\mathbb{S}$	set of schedules s, t
$\mathbb{C}$	set of configurations $\langle s, M \rangle$

Labels

ε	indicates that no rewrite occurs
σ	multiset substitution
λ	either ε or σ
$\langle\ \rangle$	empty sequence
$\overline{\sigma}$	sequence of σ labels
$\overline{\lambda}$	sequence of λ labels
$\widehat{\lambda}$	sequence of labels where all occurences of ε have been removed
$\overline{\lambda} \cdot \overline{\lambda'}$	concatenation of labels

Transition Relations

$\rightsquigarrow_1$	single-step transition of program
$\rightsquigarrow$	multi-step transition of program
$\rightsquigarrow^*$	sequence of zero or more transitions of program
$\longrightarrow_1$	single-step transition of schedule
$\longrightarrow$	multi-step transition of schedule
$\longrightarrow^*$	sequence of zero or more transitions of schedule

Notions of refinement

$\trianglelefteq$	strong statebased refinement
$\cong$	strong statebased equivalence
$\precapprox$	weak statebased refinement
$\approx$	weak statebased equivalence
$\preccurlyeq$	strong stateless refinement
$\simeq$	strong stateless equivalence
$\precsim$	weak stateless refinement
$\sim$	weak stateless equivalence
$\preccurlyeq^{\diamond}$	strong convex refinement
$=^{\diamond}$	strong convex equivalence
$\precsim^{\diamond}$	weak convex refinement
$\simeq^{\diamond}$	weak convex equivalence
$\trianglelefteq^{\phi}$	strong generic refinement
$=^{\phi}$	strong generic equivalence
$\precsim^{\phi}$	weak generic refinement
$\simeq^{\phi}$	weak generic equivalence

Bibliography

[1] J.-M. Andreoli, C. Hankin, and D. Le Métayer, editors. *Coordination Programming: Mechanisms, Models and Semantics*. Imperial College Press, 1996.

[2] K.R. Apt. *Logic Programming*, pages 493–574. Elsevier Science Publishers, 1990.

[3] S. Baase. *Computer Algorithms: Introduction to Analysis and Design*. Addison-Wesley, 1988.

[4] R.J.R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume Mathematical Centre Tracts 131. Mathematical Center, Amsterdam, 1980.

[5] R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[6] R.J.R. Back. Refinement calculus, part ii: Parallel and reactive programs. In *LNCS 430: Stepwise Refinement of Distributed Systems '89*, pages 67–93. Springer-Verlag, 1989.

[7] R.J.R. Back and R. Kurki-Suoni. Distributed cooperation with action systems. *ACM Transactions of Programming Languages and Systems*, 10(3):513–554, 1988.

[8] R.J.R. Back and R. Kurki-Suoni. Decentralization of process nets with centralized control. *Distributed Computing*, 3:73–87, 1989.

[9] R.J.R. Back and K. Sere. Stepwise refinement of action systems. In *LNCS 375: Mathematics of Program Construction '89*, pages 115–138. Springer-Verlag, 1989.

[10] R.J.R. Back and J. von Wright. Trace refinement of action systems. In *LNCS 836: CONCUR '94*, pages 368–384. Springer-Verlag, 1994.

[11] J.C.M. Baeten and P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1991.

[12] J.-P. Banâtre and D. Le Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15:55–77, November 1990.

[13] J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, January 1993.

[14] H.P. Barendrecht. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, 1981.

[15] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[16] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[17] B. Bloom, S. Istrail, and A.R. Meyer. Bisimulation can't be traced. In *Proceedings of the* 15^{th} *Symposium of Principles of Programming Languages*. ACM, 1988.

[18] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The failure of failures in a paradigm for asynchronous communication. In *LNCS 527: CONCUR'91*, pages 111–126. Springer-Verlag, 1991.

[19] S. Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127:145–163, 1996.

[20] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):46–67, 1977.

[21] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21:323–358, September 1989.

[22] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[23] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[24] M.R.V. Chaudron. Specification and verification in transaction-based parallel programming. Technical Report 92 ITI 295, TNO Institute for Applied Computer Science, TNO Institute for Applied Computer Science, March 1992. M.Sc. Thesis, Rijksuniversiteit Leiden, The Netherlands).

[25] M.R.V. Chaudron. Schedules for multiset transformer programs. Technical Report 94-36, Rijksuniversiteit Leiden, Departement of Mathematics and Computing Science, P.O. Box 9512, 2300 RA Leiden, The Netherlands, December 1994.

[26] M.R.V. Chaudron. Towards compositional design of schedules for multiset transformer programs. Technical Report 95-32, Rijksuniversiteit Leiden, Departement of Mathematics and Computing Science, P.O. Box 9512, 2300 RA Leiden, The Netherlands, November 1995.

[27] M.R.V. Chaudron. Notions of refinement for a coordination language for Gamma programs. Technical Report 96-23, Rijksuniversiteit Leiden, Departement of Mathematics and Computing Science, P.O. Box 9512, 2300 RA Leiden, The Netherlands, August 1996.

[28] M.R.V. Chaudron and A.C.N. Van Duin. A separation of concerns approach to the design of parallel algorithms for solving triangular systems of lineair equations. In *Proceedings of the ASCI'96 Conference*, pages 27–32. ASCI, Delft, 1996.

[29] M.R.V. Chaudron and E. De Jong. Notions of refinement for a coordination language for Gamma. In *Proceedings of the Theory and Formal Methods Workshop 1996*. Imperial College Press, 1996.

[30] M.R.V. Chaudron and E. De Jong. *Schedules for Multiset Transformer Programs*, pages 195–210. In Andreoli et al. [1], 1996.

[31] M.R.V. Chaudron and A.C.N. Van Duin. A method for the design of parallel algorithms a case study: Solving triangular systems. In H. El-Rewini and Y.N. Patt, editors, *Proceedings 30th annual IEEE Hawaii International Conference on Systems Science '97*, pages 320–329. Computer Society Press, 1997.

[32] M.R.V. Chaudron and A.C.N. Van Duin. The formal derivation of parallel triangular system solvers using a coordination-based design method. *Parallel Computing Journal*, (forthcoming) 1998.

[33] P. Ciancarini, D. Fogli, and M. Gaspari. *Gammalög: A Coordination Language Based on Gamma and Gödel*, pages 323–347. In Andreoli et al. [1], 1996.

[34] P. Ciancarini, R. Gorrieri, and G. Zavattaro. *An Alternative Semantics for the Calculus of GAMMA Programs*, pages 232–248. In Andreoli et al. [1], 1996.

[35] P. Ciancarini and C. Hankin, editors. *LNCS 1061: First International Conference on Coordination Languages and Models*. Springer, 1996.

[36] P. Ciancarini, K.K. Jensen, and D. Yankelevich. On the operational semantics of a coordination language. In *LNCS 924: ECOOP'94*, pages 77–106. Springer-Verlag, 1994.

[37] K.L. Clark. Parallel logic programming. *The Computer Journal*, 33(6):482–493, 1990.

[38] E.M. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *LNCS 803: A Decade of Concurrency - Reflections and Perspectives*. Springer Verlag, 1994.

[39] S. Cox, S.-H. Huang, P. Kelly, L. Liu, and F. Taylor. An implementation of statis functional process networks. In *LNCS 605: PARLE'92*, pages 479–512. Springer-Verlag, 1992.

[40] J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978.

[41] J. Darlington. *Program Transformation*, pages 193–215. Cambridge University Press, 1982.

[42] J. Darlington and L. While. Controlling the behaviour of functional language systems. In *LNCS 274: Proceedings Functional Programming Languages and Computer Architecture 1987*, pages 278–299. Springer-Verlag, 1987.

[43] B.A. Davey and H.A. Priesty. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[44] R. De Nicola and R. Pugliese. *A Process Algebra based on Linda*, pages 160–178. In Ciancarini and Hankin [35], 1996.

[45] R. De Nicola and R. Pugliese. Testing semantics of asynchronous distributed programs. In *LNCS 1192: Analysis and Verification of Multiple Agent Languages*, pages 320–344. Springer-Verlag, 1997.

[46] R. de Simone. Higher-level synchronising devices in MEIJE-SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[47] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[48] J. Dingel. Modular verification for shared-variable concurrent programs. In *LNCS 1119: Proceedings CONCUR 1996*, pages 703–719. Springer-Verlag, 1996.

[49] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, 1991.

[50] G. Dromey. *Program Derivation: The Development of Programs from Specifications*. Addison-Wesley, 1989.

[51] L. Errington, C. Hankin, and T.P. Jensen. Reasoning about Gamma programs. In *Proceedings of the Imperial College Theory and Formal Methods Workshop 1993*, pages 115–125. Imperial College Press, 1993.

[52] L. Errington, C. Hankin, and T.P. Jensen. A congruence for Gamma programs. In *LNCS 724: Static Analysis*. Springer-Verlag, 1996.

[53] P. Fradet and D. Le Métayer. Structured Gamma. Technical Report 989, INRIA-Rennes, INRIA, Campus Universitaire de Beaulieu, 35042 - RENNES CEDEX FRANCE, March 1996.

[54] P. Fradet and D. Le Métayer. Shape types. In *Proceeding of* 24^{th} *ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages 1997*. ACM Press, January 1997.

[55] P. Fradet and D. Le Métayer. Type checking for a multiset rewriting language. In *LNCS 1192: Analysis and Verification of Multiple Agent Languages 1997*, pages 126–140. Springer-Verlag, 1997.

[56] K.A. Gallivan, R.J. Plemmons, and A.H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, March 1990.

[57] S.J. Gay and C. Hankin. *A Program Logic for Gamma*, pages 195–210. In Andreoli et al. [1], 1996.

[58] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[59] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, January 1987.

[60] C. Green and D. Barstow. On program synthesis knowlegde. *Artificial Intelligence*, 10:241–279, 1978.

[61] D. Gries. *The Science of Computer Programming*. Springer-Verlag, 1981.

[62] D. Gries and J. Misra. A linear algorithm for finding prime numbers. *Communications of the ACM*, 21(12):999–1003, December 1978.

[63] J.F. Groote and A. Ponse. Process algebra with guards: Combining Hoare logic with process algebra. *Formal Aspects of Computing*, 6(2):115–164, Mar-Apr 1994.

[64] J.F. Groote and F. Vaandrager. Structured operational semantics and bisimulation as a congruence. In *LNCS 372: ICALP '89*, pages 423–438. Springer-Verlag, 1989.

[65] C. Hankin, D. Le Métayer, and D. Sands. A calculus of Gamma programs. In *LNCS 757: 5^{th} International Workshop on Languages and Compilers for Parallel Computing*, pages 342–355. Springer-Verlag, 1992.

[66] C. Hankin, D. Le Métayer, and D. Sands. A calculus of Gamma programs. Technical Report 674, INRIA-Rennes, INRIA, Campus Universitaire de Beaulieu, 35042 - RENNES CEDEX FRANCE, July 1992.

[67] C. Hankin, D. Le Métayer, and D. Sands. A parallel programming style and its algebra of programs. In *LNCS 694: PARLE '93*, pages 367–378. Springer-Verlag, 1993.

[68] R. Harrison and H. Glaser. The Gamma model as a functional programming tool. In *LNCS 468: Advances in Computing and Information*, pages 164–173. Springer-Verlag, 1990.

[69] P.J. Hayes. Computation and deduction. In *Proceedings of the 2nd Symposium on Mathematical Foundations of Computer Science*, pages 105–118. Czechoslovak Academia of Sciences, 1973.

[70] E. Hedman, J.N. Kok, and K. Sere. Coordinating action systems. In *Proceedings of the 1997 Coordination conference, Berlin (to be published in the LNCS series)*, 1997.

[71] P. Hill and J. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[72] C.A.R. Hoare. Quicksort. *BCS Computer Journal*, 5(1):10–15, 1962.

[73] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[74] C.A.R. Hoare. Proof of a structured program: The sieve of Erathostenes. *BCS Computer Journal*, 15(4):321–325, November 1972.

[75] C.A.R. Hoare. Parallel programming: An axiomatic approach. *Computer Languages*, 1(2):151–160, June 1975.

[76] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[77] P. Hoogendijk. (relational) programming in the Boom hierarchy of types. In *LNCS 669: Mathematics of Program Construction '92*, pages 163–190. Springer-Verlag, 1993.

[78] P. Hudak. Exploring parafunctional programming: Separating the what from the how. *IEEE Software*, January 1988.

[79] E. de Jong. *Transaction-based Programming*. PhD thesis, Leiden University, The Netherlands, Departement of Mathematics and Computing Science, December 1992.

[80] A. Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice-Hall, 1990.

[81] D. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley, 1973.

[82] R. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.

[83] R. Kowalski. *Logic for Problem Solving*. Elsevier Science Publishers, 1979.

[84] R. Kowalski. *Logic as a Computer Language*, pages 3–16. Academic Press, 1982.

[85] L. Lamport. The "Hoare logic" of concurrent programs. *Acta Informatica*, 14:21–37, 1980.

[86] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[87] J. Lloyd. *Foundations of Logic Programming*. Springer, 1987.

[88] L.D.J.C. Loyens and R.H. Bisseling. The formal construction of a parallel triangular system solver. In *LNCS 669: Mathematics of Program Construction 1989*, pages 325–334. Springer-Verlag, 1989.

[89] H. McEvoy. *Gamma, Chromatic Typing and Vegetation*, pages 368–387. In Andreoli et al. [1], 1996.

[90] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[91] R. Milner. Functions as processes. In *LNCS 443: Proceedings of ICALP'90*, pages 167–180. Springer-Verlag, 1990.

[92] E. Nöcker, J. Smetsers, M. Eekelen, and M. Plasmeijer. Concurrent clean. In *LNCS 506: PARLE '91, Volume II*, pages 202–219. Springer-Verlag, 1991.

[93] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(1):319–340, 1976.

[94] D.M.R. Park. Concurrency and automata on infinite sequences. In *LNCS 104: Proceedings of the 5th G. I. Conference on Theoretical Computer Science*, pages 167–183. Springer-Verlag, 1980.

[95] A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 2(28):360–414, 1996.

[96] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Dept. of Computer Science, Aarhus University, Ny Munkegade, Building 540 DK-8000 Aarhus C, Denmark, September 1981. Reprint April 1991.

[97] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18^{th} Symposium on the Foundations of Computer Science*. ACM, November 1977.

[98] K.V.S. Prasad. Programming with broadcasts. In *LNCS 715: CONCUR'93*, pages 173–201. Springer-Verlag, 1993.

[99] K.V.S. Prasad. A calculus of broadcasting systems. *Science of Computer Programming*, 25:285–327, 1995.

[100] V. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[101] P. Pritchard. Linear prime-number sieves: A family tree. *Science of Computer Programming*, 9:17–35, 1987.

[102] M.J. Quinn. *Parallel Computing: Theory and Practice (2^{nd} ed).* McGraw-Hill, 1994.

[103] M. Reynolds. *Temporal Semantics for Gamma*, pages 141–170. Volume Coordination Programming: Mechanisms, Models and Semantics of Andreoli et al. [1], 1996.

[104] C.H. Romine and J.M. Ortega. Parallel solution of triangular systems of equations. *Parallel Computing*, 6:109–114, 1988.

[105] D. Sands. A compositional semantics of combining forms for Gamma programs. In *LNCS 735: Formal Methods in Programming and Their Applications*, pages 43–56. Springer-Verlag, 1993.

[106] D. Sands. Laws of synchronised termination. In *Proceedings of the First Imperial College Departement of Computing Workshop on Theory and Formal Methods*, pages 276–288. Springer-Verlag, 1993.

[107] D. Sands and M. Weichert. From Gamma to CBS: Refining multiset transformations with broadcasting processes. In *Proceedings 31th annual IEEE Hawaii International Conference on Systems Science '98*. Computer Society Press, 1998. to be published.

[108] K. Sere. *Stepwise Derivation of Parallel Algorithms*. PhD thesis, øAbo Akademi, Departement of Computer Science, Finland, September 1990.

[109] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[110] A.K. Singh. Program refinement in fair transition systems. *Acta Informatica*, 30:503–535, 1993.

[111] R.T. Udink. *Program Refinement in UNITY-like Environments*. PhD thesis, Utrecht University, The Netherlands, 1995.

[112] M. Weichert. A note on Chaudron and De Jong's schedules for Gamma programs. Private Communication, November 1997.

[113] J.C.P. Woodcock and C. Morgan. Refinement of state-based concurrent systems. In *LNCS 428: VDM'90 – Formal Methods in Software Development*, pages 340–351. Springer-Verlag, 1990.

List of Figures

Index

U

Samenvatting

Het Scheiden van Berekening en Coordinatie bij het Ontwerp van Parallelle en Gedistribueerde Programma's

In dit hoofdstuk worden de belangrijkste concepten van dit proefschrift beschreven. Getracht is om deze uitleg voor een breed publiek toegankelijk te maken. In de volgende secties leggen we achtereenvolgens de begrippen "Parallel Rekenen", "Coördinatie" en "Formele Methoden" uit. Aan de hand van de uitleg van deze begrippen beschrijven we de bijdrage van het onderzoek dat is beschreven in dit proefschrift.

Parallel Rekenen

Het merendeel van de huidige generatie computers is gebaseerd op de zogenaamde Von Neumann architectuur. Deze architectuur bestaat globaal uit drie componenten: een geheugen waarin een programma kan worden opgeslagen, een geheugen waarin data waarmee het programma werkt, kan worden opgeslagen, en een verwerkingseenheid (processor). Als een programma en de bijbehorende data in de betreffende geheugens geplaatst zijn, werkt deze architectuur volgens de volgende procedure:

1. De verwerkingseenheid haalt een instructie uit het programma-geheugen en gegevens uit het data-geheugen.
2. De instructie wordt op de gegevens toegepast, en het resultaat wordt in het data-geheugen opgeslagen.
3. De verwerkingseenheid bepaalt wat de volgende instructie is, en herhaalt deze procedure vanaf stap 1.

De verbetering van de verwerkingssnelheid van computers die in de afgelopen jaren is geboekt, is vooral te danken aan de versnelling van de individuele stappen uit deze

procedure. Alhoewel deze snelheid nog steeds verbeterd wordt, is er een fysische grens aan de maximaal haalbare snelheid.

Bij de verwerking van een programma op een Von Neumann-architectuur wordt op ieder moment precies één instructie uitgevoerd. De verwerking van een programma zou versneld kunnen worden door meerdere instructies van dat programma gelijktijdig door verschillende verwerkingseenheden uit te laten voeren. Stel dat we een programma in twee helften kunnen splitsen en de instructies van deze programma-delen door verschillende verwerkingseenheden uit kunnen laten voeren. Dan kan dit programma door twee verwerkingseenheden twee maal zo snel verwerkt worden als door één enkele verwerkingseenheid (zonder dat de verwerkingssnelheid van individuele verwerkingsheden verhoogd is).

In principe geldt, dat als een programma (gelijkmatig) over n verwerkingseenheden verdeeld kan worden, het programma n maal zo snel verwerkt kan worden dan dat het verwerkt kan worden door een enkele verwerkingseenheid. Het principe van het verwerken van een programma door meerdere verwerkingseenheden staat binnen de Informatica bekend als *parallel rekenen*.

Door de verregaande miniaturisering en prijsverlaging van individuele processoren is het realistisch geworden om systemen met hoge verwerkingssnelheden te bouwen door deze samen te stellen uit meerdere processoren. In dit geval spreekt men over parallelle computersystemen. Momenteel bestaan er parallelle computers met duizenden processoren.

Zoals gezegd kan de verwerkingstijd van een programma verkort worden indien dat programma beschreven kan worden als een verzameling taken waaraan gelijktijdig gerekend kan worden. Echter, volledig gelijktijdige uitvoering van taken is alleen mogelijk indien deze taken onafhankelijk zijn. Omdat deze taken doorgaans samenwerken aan het oplossen van één probleem, zijn ze vaak juist afhankelijk van elkaar. Zo'n afhankelijkheid kan zich bijvoorbeeld manifesteren in de vorm van verschillende taken die gelijktijdig een bewerking op eenzelfde gegeven uit willen voeren. In dit geval moet ieder van deze taken op zijn beurt wachten. Een ander soort afhankelijkheid doet zich voor als een taak een berekening uit wil voeren waarvoor het een resultaat van een andere taak nodig heeft. In zulke gevallen moet de uitvoering van de eerstgenoemde taak wachten tot het resultaat van de andere taak beschikbaar is.

Bij het ontwerpen van programma's voor parallelle computers wordt getracht om een oplossingsmethode te bedenken die opgedeeld kan worden in zo onafhankelijk mogelijke taken. Vervolgens wordt getracht deze taken zodanig over de beschikbare processoren

te verdelen dat de hierboven beschreven problemen (gelijktijdig gebruik van dezelfde gegevens en wachten op resultaten van andere taken) de verwerkingstijd zo min mogelijk vertragen.

Het ontwerpen van programma's die voorschrijven hoe tientallen, honderden of zelfs duizenden processoren samen een taak moeten verwerken, stelt software ingenieurs voor grote problemen. In dit proefschrift wordt een methode voorgesteld die het ontwerpen van dergelijke programma's vereenvoudigt. In de volgende secties zullen we dieper ingaan op de principes die aan deze methode ten grondslag liggen.

Coördinatie

In de wereld om ons heen spelen veel processen zich gelijktijdig af. In sommige gevallen moeten we bij gelijktijdige processen ingrijpen om problemen te voorkomen. Denk bijvoorbeeld aan het voorkomen van gelijktijdig gebruik van een kruispunt door verschillende voertuigen. In zulke gevallen schakelen we een proces in dat het gedrag van de individuele processen aan-/bijstuurt (in het voorbeeld: een verkeersregelinstallatie). Dit proces coördineert het samenspel van de andere processen op zodanige wijze dat alle betrokken processen hun taken kunnen uitvoeren ook al hebben ze strijdige belangen.

In dit proefschrift wordt een methode voor het ontwerpen van parallelle programma's beschreven die gebaseerd is op het idee dat dit voor een deel een coordinatie-probleem is. Er moet namelijk antwoord gegeven worden op de vraag "Wanneer kan welke deeltaak van de oplossingsmethode uitgevoerd worden?".

De methode voor het ontwerpen van programma's die in dit proefschrift beschreven wordt, stelt de volgende fasering voor.

1. In de eerste fase dient een oplossingsmethode ontworpen te worden. Deze oplossingsmethode wordt beschreven door aan te geven uit welke deeltaken zij bestaat.

   Het Gamma-model (uit Hoofdstuk 2) ondersteunt de beschrijving van een oplossingsmethode in termen van deeltaken. Met behulp van deze wijze van specificeren is het mogelijk om (wiskundig) te redeneren over de correctheid van een oplossingsmethode terwijl de beschrijving hiervan onafhankelijk blijft van een specifieke manier van uitvoeren.

2. In de tweede fase kunnen één of meer strategieën ontworpen worden die vastleggen op welke wijze de deeltaken van de oplossingsmethode moeten worden uitgevoerd.

Deze strategieën dienen gescheiden van de oplossingsmethode beschreven te worden in termen van een coördinatie-taal. In Hoofdstuk 3 wordt een coördinatie-taal ontwikkeld die aansluit bij het Gamma-model van Hoofdstuk 2.

De uitvoeringsstrategieën van een oplossingsmethode kunnen worden toegespitst op bepaalde eigenschappen van een computersysteem (bijvoorbeeld het aantal beschikbare processoren). Doordat de uitvoeringsstrategie gescheiden van de oplossingsmethode wordt beschreven, is het relatief eenvoudig een oplossingsmethode geschikt te maken voor verschillende typen computersystemen. Hiervoor hoeft namelijk slechts de uitvoeringsstrategie aangepast te worden.

Deze methode voor de ontwikkeling van programma's dwingt de ontwerper de correctheids-aspecten en efficiency-aspecten van zijn programma in verschillende fases aan te pakken. Als gevolg hiervan kan in ieder van deze fases geabstraheerd worden van de aspecten die thuis horen in de andere fase. Deze fasering is een verbetering ten opzichte van conventionele methoden waarin de ontwerper makkelijk wordt verleid tot het tegelijkertijd oplossen van correctheids- en efficiency-problemen.

Voor zowel de eerste als tweede fase van deze ontwerpmethode worden in dit proefschrift wiskundige – "formele" – methoden beschreven die de ontwerper in staat stellen zijn ontwerp op ondubbelzinnige wijze te beschrijven en hierover te redeneren. Het gebruik van deze methoden kan helpen bij het voorkomen van fouten in de ontworpen programma's. In de volgende sectie gaan we dieper in op deze "formele methoden".

Formele Methoden

Voordat een programma in gebruik genomen wordt, dient vertrouwen te worden verkregen in de juistheid van de werking van dit programma; m.a.w. er dient nagegaan te worden of het programma in alle situaties die zich voor kunnen doen, het bedoelde resultaat levert. Deze eigenschap wordt de *correctheid* van een programma genoemd.

Een eerste probleem dat bij het vaststellen van de correctheid van een programma vaak opgelost moet worden, is dat er geen duidelijkheid is omtrent de bedoeling van het programma. Hierdoor ontbreekt een criterium waaraan de werking van een programma getoetst kan worden. Een veel gebruikte manier om hiermee om te gaan is om de eisen die aan het programma gesteld worden, op meer of minder gestructureerde wijze, in natuurlijke taal, op te schrijven. Dergelijke beschrijvingen leiden echter vaak aan vaagheid, onvolledigheid en dubbelzinnigheid.

Desalniettemin wordt met de houvast die zo'n beschrijving biedt, overgegaan naar een volgende fase waarin getracht wordt te verifiëren of een programma aan de gestelde eisen voldoet. Hiertoe wordt getest of het uitvoeren van het programma voor een verzameling van (begin)situaties de gewenste resultaten levert. Het aantal situaties waarin een programma kan verkeren is doorgaans echter vele ordes groter dan het aantal situaties dat door middel van testen binnen redelijke tijd gecontroleerd kan worden.

Een aanpak die belooft een significante bijdrage te kunnen leveren aan het produceren van correcte programma's is het gebruik van wiskundige methoden. Met behulp van wiskundige methoden is het mogelijk om een precieze, ondubbelzinnige en volledige beschrijving van de eisen aan en werking van een programma op te stellen. Bovendien kunnen op een wiskundige beschrijving van een programma wiskundige methoden van redeneren toegepast worden. Hierdoor kan (in wiskundige zin) bewezen worden dat programma's voldoen aan bepaalde eigenschappen. De verzameling van wiskundige theorieën voor het modelleren van en redeneren over programma's wordt "formele methoden" genoemd.

De methode voor het ontwikkelen van programma's die in dit proefschrift wordt voorgesteld, behelst een bouwwerk van formele methoden. Het fundament hiervan bestaat uit wiskundige definities van de syntax (vorm) en semantiek (betekenis) van de talen voor het specificeren van oplossingsmethoden en uitvoeringsstrategieën (Hoofdstukken 2 en 3). Op basis van deze definities wordt een formele methode voor redeneren over oplossingsmethoden gedefiniëerd. Daarnaast maken deze definities het mogelijk om specificaties van oplossingsmethoden en uitvoeringsstrategieën aan elkaar te relateren. Hierdoor is het mogelijk om voor iedere oplossingsmethode een generieke uitvoeringsstrategie te construeren.

De volgende fase van de ontwikkelmethode bestaat uit een verzameling technieken voor het toespitsen van een uitvoeringsstrategie op de bijbehorende oplossingsmethode. Hiervoor wordt het begrip "verfijning" geïntroduceerd. Een uitvoeringsstratgie is een verfijning van een andere uitvoeringsstratgie indien de eerste op meer specifieke wijze vastlegt hoe een oplossingmethode moet worden uitgevoerd dan de tweede. De theorie van verfijning wordt geïntroduceerd in Hoofdstuk 4. Hierin wordt geïllustreerd hoe eigenschappen van een oplossingsmethode kunnen worden gebruikt voor het verfijnen van de uitvoeringsstrategie.

Vanuit praktisch oogpunt is het wenselijk om uitvoeringsstrategieën op een modulaire manier te kunnen verfijnen; d.w.z. het verfijnen van een deel van een uitvoeringsstrategie moet leiden tot een verfijning van de gehele uitvoeringsstrategie. Om deze modulaire

manier van verfijning formeel te rechtvaardigen moet in de definitie van verfijning rekening gehouden worden met het feit dat een uitvoeringsstrategie deel uit kan maken van een omvattende uitvoeringsstrategie.

Om een verfijning in zo veel mogelijk situaties toe te kunnen passen moet de notie van verfijning met zo veel mogelijk omvattende uitvoeringsstrategieën rekening houden. De relatie die bestaat tussen uitvoeringsstrategieën en oplossingsmethoden impliceert echter dat er dan ook met veel mogelijke oplossingsmethoden rekening gehouden wordt. En met hoe meer oplossingsmethoden rekening gehouden wordt, des te minder gemeenschappelijke eigenschappen deze hebben waarop een uitvoeringsstrategie toegespitst kan worden. De kunst van het definiëren van een bruikbare definitie van verfijning komt hierom neer op het kiezen van een verzameling omvattende uitvoeringsstrategieën die een balans slaat tussen algemeenheid (voor algemene toepasbaarheid) en specifiekheid (om eigenschappen te houden om uitvoeringsstrategieën op toe te spitsen).

Ter ondersteuning van deze keuze wordt in Hoofdstuk 5 een generieke theorie van verfijning ontwikkeld. Deze theorie identificeert eigenschappen waaraan een verzameling van omvattende uitvoeringsstrategieën moet voldoen om te garanderen dat de corresponderende definitie van verfijning op modulaire wijze gebruikt mag worden.

Op basis van deze generieke theorie wordt in Hoofdstuk 6 een nieuwe variant van verfijning voorgesteld. Analyze van deze variant toont aan dat ze de voornaamste gunstige eigenschappen van de eerder (in Hoofdstuk 4) onderzochte varianten van verfijning combineert. Tevens wordt met behulp van de generieke theorie van verfijning aangetoond dat het mogelijk is om de verschillende varianten van verfijning te combineren. Hierdoor is er voor de ontwikkeling van uitvoeringsstrategieën een arsenaal van verfijningsmethoden beschikbaar.

Uiteindelijk worden in Hoofdstuk 7 een aantal case-studies uitgevoerd. Deze tonen aan dat de voorgestelde methode voor het ontwikkelen van programma's gerealiseerd kan worden met behulp van de in dit proefschrift ontwikkelde formele methoden.

Curriculum Vitae

Michel Chaudron is geboren op 1 augustus 1969 te Leiden. In 1987 behaalde hij het VWO diploma aan het Rijnlands Lyceum te Wassenaar waarna hij Informatica ging studeren aan de Rijksuniversiteit te Leiden. In augustus 1988 behaalde hij het propaedeutisch examen.

De eerste helft van zijn vierde studiejaar (september 1990 - maart 1991) bracht hij door bij de Programming Research Group van de Universiteit van Oxford. Daar volgde hij de colleges van de Master of Science cursus in Computation en werkte aan een tweetal research projecten op het gebied van formele methoden. Na zijn terugkomst in Nederland startte hij in september 1991 zijn afstudeerstage bij het (toenmalige) Instituut voor Toegepaste Informatica van TNO Delft. Tot maart 1992 werkte hij daar aan een afstudeerproject op het gebied van ontwerp en verificatie van parallelle algoritmes.

Van juni 1992 tot juni 1997 werkte hij als AIO aan het onderzoek waarvan de resultaten zijn beschreven in dit proefschrift. Gedurende deze periode was hij van september 1994 tot en met december 1994 als visiting researcher op bezoek bij de Theory and Formal Method Section van Imperial College in London. Naast zijn onderzoeksactiviteiten maakte Michel enkele jaren deel uit van de organisatie van het Graduate Network for Applied Research in Parallel Computing.

Sinds juni 1997 werkt hij bij een automatiseringsbedrijf.